UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

UMA FERRAMENTA DE ANÁLISE DE ROBUSTEZ PARA A MELHORIA DA QUALIDADE DE SISTEMAS DE SOFTWARE

ISRAEL BARBOSA GARCIA

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal do Rio Grande do Norte como parte dos requisitos necessários para obtenção de grau de Mestre em Engenharia de Produção.

MESTRE EM ENGENHARIA DE PRODUÇÃO

Natal/RN
2013

ISRAEL BARBOSA GARCIA

UMA FERRAMENTA DE ANÁLISE DE ROBUSTEZ PARA A MELHORIA DA QUALIDADE DE SISTEMAS DE SOFTWARE

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal do Rio Grande do Norte como parte dos requisitos necessários para obtenção de grau de Mestre em Engenharia de Produção.

Orientador: Nélio Cacho, Dr.

Natal/RN
2013

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Garcia, Israel Barbosa.
Uma ferramenta de análise de robustez para a melhoria da qualidade de sistemas de software. / Israel Barbosa Garcia. – Natal, RN, 2013.
91 f.; il.

Orientador: Prof. Dr. Nélio Cacho.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Tecnologia. Programa de Pós-Graduação em Engenharia de Produção.

1. Sistemas de software – Dissertação. 2. Sistemas de software – Qualidade – Dissertação. 3. Tratamento de exceções – Softwares – Dissertação. 4. Robustez – Dissertação. 5. Sistemas de software - Análise estática – Dissertação. 6. NET – Dissertação. I. Cacho, Nélio. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU 004:681.5

ISRAEL BARBOSA GARCIA

UMA FERRAMENTA DE ANÁLISE DE ROBUSTEZ PARA A MELHORIA DA QUALIDADE DE SISTEMAS DE SOFTWARE

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal do Rio Grande do Norte como parte dos requisitos necessários para obtenção de grau de Mestre em Engenharia de Produção, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. Nélio Alessandro Azevedo Cacho
Presidente da Banca (Orientador)
Universidade Federal do Rio Grande do Norte - UFRN

Prof. Dr. Luciano Ferreira
Membro Interno
Universidade Federal do Rio Grande do Norte - UFRN

Prof. Dr. Gilbert Azevedo da Silva
Membro Externo
Instituto Federal de Educação, Ciência e Tecnologia - IFRN

Natal-RN, 31 de Janeiro de 2013.

Dedico este trabalho a Deus, acima de tudo, aos meus pais Elias (in memoriam) e Graça, pela dedicação e exemplo de vida, à minha esposa Raissa e filha Raquel, que me apoiaram e incentivaram em todos os momentos dessa jornada.

AGRADECIMENTOS

A Deus pela vida.

A meu pai Professor Elias (in memoriam) que desde muito cedo é exemplo de dedicação aos estudos.

A minha mãe Graça que é exemplo de paciência.

A minha esposa Raissa que paciência nos momentos que tive que me ausentar para me dedicar aos estudos.

A minha filha Raquel que é minha razão de viver e que do seu jeito me dava forças para continuar.

A meus irmãos Daniel, Elias Junior e Gabriel pelo apoio moral.

Ao meu orientador e amigo Nélio Cacho por acreditar no meu potencial e pela paciência.

Aos amigos Nicholas Paiva, Eliezio Soares e Andre Castro pelo apoio intelectual e incentivo.

Ao Dr. Hematologista Aldair Paiva que me tranquilizou e me tratou durante o grave problema de saúde que tive durante o mestrado.

Por fim, a todos aqueles que, direta ou indiretamente, contribuíram para a realização deste trabalho.

“No fim tudo da certo. Se não deu certo é por que não chegou ao fim.”
Fernando Sabino

RESUMO

Um produto de software é considerado confiável quando ele consegue entregar suas funcionalidades da forma como elas foram definidas. *Robustez* é um sub-atributo de confiabilidade e diz respeito à capacidade do software em reagir especificamente a defeitos externos. Os mecanismos de tratamento de exceções deveriam garantir a robustez dos sistemas. Entretanto, na prática é difícil atingir tal objetivo, seja por mau uso dos modelos existentes, seja pela deficiência dos próprios modelos. Além disso, percebe-se que os desenvolvedores que tratam as exceções de maneira adequada acabam tendo problemas sérios de produtividade, ao passo que o seu negligenciamento embora seja mais produtivo, tende a gerar sistemas menos confiáveis e com subseqüente aumento do custo de pós-produção. Alguns desastres reais tiveram relação direta com o negligenciamento do tratamento de exceções, por exemplo: o caso do foguete europeu Ariane 5 que se auto-destruiu logo após o lançamento, e do sistema do radar brasileiro X-4000 que foi indicado como uma das causas do acidente aéreo do vôo GOL 1907 em 2006. Nesse contexto, este trabalho apresenta a ferramenta de análise estática do fluxo excepcional eFlowMining, focada na melhoria da robustez de aplicações .NET. Ela permite que o desenvolvedor: visualize métricas coletadas sobre o comportamento excepcional; analise o fluxo excepcional através de uma representação gráfica em forma de árvore; identifique possíveis bugs entre diferentes versões da mesma aplicação; e localize de forma rápida os tipos de exceções lançadas e seus respectivos tratadores. Todas as informações coletadas são armazenadas em bancos de dados a fim de possibilitar consultas e comparações das análises realizadas. A avaliação da ferramenta foi dividida em duas fases. A primeira teve o objetivo de mostrar a compatibilidade e a precisão da ferramenta em relação às diferentes linguagens de programação suportadas pela plataforma .NET. A segunda avaliou como a ferramenta ajudou os desenvolvedores a identificar possíveis defeitos entre diferentes versões do mesmo sistema de software.

Palavras-chave: Sistemas de Software. Qualidade. Tratamento de Exceções. Robustez. Análise Estática. Plataforma .NET.

ABSTRACT

A software product is considered reliable if it can deliver its functions the way they were defined. Robustness is a sub-attribute of reliability and concerns the software's ability to respond specifically to external defects. The exception handling mechanisms should ensure the robustness of the systems. However, in practice it is difficult to achieve such a goal, either by misuse of existing models, either by deficiency of the models themselves. Moreover, it is clear that developers dealing exceptions properly end up having serious problems in productivity, while its neglect although more productive, tends to generate less reliable systems and subsequent rising cost of post-production. Some real disasters were directly related to the neglect of exception handling, for example: the case of the European Ariane 5 rocket that self-destructed soon after launch, and the radar system Brazilian X-4000 which was nominated as one of the causes the crash of Gol Flight 1907 in 2006. In this context, this work presents a static analysis tool exceptional flow eFlowMining, focused on improving the robustness of applications. NET. It allows the developer: view metrics collected on the exceptional behavior; analyze the exceptional flow via a graphical representation as a tree; identify possible bugs between different versions of the same application, and quickly locate the types of exceptions thrown and their their handlers. All information collected is stored in databases to enable searches and comparisons of the analyzes. The evaluation tool was divided into two phases. The first aimed to show the consistency and accuracy of the tool relative to the different programming languages supported by the platform. NET. The second assessed how the tool helped developers identify possible defects between different versions of the same software system.

Keywords: Software System. Quality. Exception Handling. Reliability. Static Analysis. Platform .NET.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CCI - Common Compiler Infrastructure

EH - Exception Handling

EHC - Exception Handling Context

GUI - Interface Gráfica com o Usuário

IEEE - Institute of Eletrical and Eletronic Engineers

OO – Orientação a objetos

RAIL - Runtime Assembly Instrumentation Library

V&V – Validação e verificação

SUMÁRIO

1 INTRODUÇÃO

Atualmente todos os setores da sociedade demandam sistemas de software. Praticamente todas as atividades produtivas da indústria são automatizadas, o setor de serviços depende dos sistemas para se comunicar, vender, controlar, gerenciar; atividades do setor primário tais como, agricultura e mineração, também usufruem das vantagens competitivas da automação. Sendo assim, a produção e manutenção de software de qualidade dentro de custos e prazos apropriados é uma tarefa importante para o funcionamento da economia.

Nesse contexto, a área de produção das empresas de software precisa conhecer seu papel dentro da estratégia corporativa. Segundo Slack, Chambers e Johnston (2002), a função produção de qualquer tipo de empresa pode ter o papel de ser apoio para a estratégia empresarial, desenvolvendo objetivos e políticas apropriadas aos recursos que a mesma administra; implementadora da estratégia empresarial, transformando as decisões estratégicas em realidade operacional; ou como impulsionadora da estratégia empresarial, fornecendo meios para a obtenção de vantagem competitiva. Tal vantagem pode ser alcançada através de cinco objetivos básicos de desempenho: qualidade, rapidez, credibilidade, flexibilidade e custo; que são aplicados a todos os tipos de operações produtivas.

Cada objetivo de desempenho tem sua definição que pode ser sintetizada da seguinte forma: qualidade, fazer certo; rapidez, entregar no tempo certo; credibilidade, cumprir prazos; flexibilidade, mudar a operação de acordo com a necessidade; e custo, proporcionar competitividade e margem de lucro. O significado desses objetivos varia para cada tipo de operação produtiva, por exemplo: qualidade para uma empresa de ônibus pode significar ônibus limpos, arrumados e silenciosos; já uma empresa de software leva em conta alguns fatores, entre eles, usabilidade, confiabilidade, funcionalidade, etc. Dentre os cinco objetivos, o que mais é afetado pela forma como os outros são conduzidos é o custo. A produtividade gerada por uma operação com qualidade, rápida, flexível e com credibilidade resultará em redução de custos, aumentando assim a competitividade (SLACK; CHAMBERS; JOHNSTON, 2002).

No contexto das empresas de software, o objetivo qualidade tem um destaque importante. Estima-se que o custo de pós-produção de software equivale a 70% do orçamento total (BOEHM, 1979). O desenvolvimento de software com qualidade reduz esse custo de pós-produção, satisfaz as exigências do mercado, além de resultar em produtos confiáveis.

Um produto de software é considerado confiável quando ele consegue entregar suas funcionalidades da forma como elas foram definidas. Lee e Anderson (1990) definiram *Confiabilidade* como sendo a probabilidade de operação livre de falhas durante um período especificado, ou seja, a confiabilidade de um software deve ser inversamente proporcional a quantidade de falhas. Sendo assim, um produto de software confiável precisa tolerar a manifestação de falhas evitando que o mesmo sofra desvios em sua especificação. *Robustez* é um sub-atributo de confiabilidade e diz respeito à capacidade do software em reagir especificamente a defeitos externos, ou seja, defeitos originados fora do contexto do sistema (BOEHM; BROWN; LIPOW, 1976; LAPRIE; RANDELL, 2004); por exemplo, interação errada com o sistema por parte dos usuários, entradas em não-conformidade com as especificações, defeitos de hardware, etc. O desenvolvimento de um software robusto requer uma complexidade adicional, principalmente nos mecanismos de detecção e tratamento dos defeitos, bem como no controle de fluxo das situações normais e excepcionais, afim de que não ocorram falhas que impossibilitem o uso do sistema. A abordagem mais utilizada para resolver os problemas gerados por essa complexidade adicional é o mecanismo de tratamento de exceções.

Os mecanismos de tratamento de exceções (GOODENOUGH, 1975) são utilizados no desenvolvimento de software a fim de alcançar níveis aceitáveis de robustez, estruturando o fluxo excepcional dos sistemas em todas as fases do ciclo de vida. Tais mecanismos também provêem facilidade de entendimento, manutenção e reuso através da separação explicita entre o código normal e o excepcional (GARCIA *et al*, 2001). Essa separação garante que alterações no comportamento normal não afetem o comportamento excepcional e vice-versa, bem como ratifica a propriedade de modularidade do software (PARNAS; WURGES, 1976). Várias linguagens de programação orientadas a objetos, tais como, Java, C++, C# e VB.NET, implementam de forma nativa esses mecanismos, através de construções que indicam a ocorrência de um erro e sua respectiva associação às ações para a recuperação do problema. No entanto, a forma como esse suporte é materializado incorre em alguns problemas (GARCIA, 2001), principalmente relacionados à manutenibilidade do software, tais como: código espalhado, código de atividade normal misturado com código de atividade excepcional, dificuldade de reuso e fluxo excepcional global implícito (MALAYERI; ALDRICH, 2006; CACHO *et al*, 2008).

Nesse contexto, também é importante citar o paradigma de projeto Orientado a Objetos (OO) que tem como objetivos gerais (BOOCH, 1994): abstração, desenvolvimento

evolutivo, especialização funcional e modelagem conceitual. Tais objetivos são suportados nas linguagens de programação através de elementos básicos que permitem o encapsulamento, modularidade, especialização e o reuso através da herança (generalização) (MILLER; TRIPATHI, 1997). Embora seja a abordagem mais utilizada para se desenvolver software de qualidade, a combinação da metodologia orientada a objetos com os mecanismos de tratamento de exceções nem sempre ocorre de forma satisfatória, pois alguns conflitos podem surgir (MILLER; TRIPATHI, 1997), como: definição das interfaces dos objetos e dos relacionamentos de composição, contexto e conformidade das exceções, e controle do fluxo do sistema, tais conflitos geralmente se manifestam durante a especificação de requisitos e na evolução do projeto.

Em virtude dos problemas citados, as empresas de software necessitam de processos e ferramentas que as ajudem a garantir níveis aceitáveis de robustez de seus produtos. Atividades de validação e verificação (V&V) são utilizadas para esse fim, elas visam assegurar que tanto o modo como o software está sendo desenvolvido quanto o produto em si estejam em conformidade com a especificação e com as expectativas dos usuários. Embora sejam termos parecidos, eles possuem uma sucinta diferença. Boehm (1979) expressou essa diferença ao definir que a atividade de validação visa garantir que o produto atenda as expectativas dos usuários, já a atividade de verificação visa assegurar que o produto esteja em conformidade com as especificações.

A análise estática de código é umas das técnicas utilizadas nas atividades de validação e verificação, ela consiste da extração de informações semânticas sobre um software em tempo de compilação (LANDI, 1992). Essa extração é realizada por ferramentas automatizadas que examinam o software sem executá-lo, visando determinar propriedades do produto válidas para qualquer execução final, localizar defeitos, bem como ajudar no entendimento das estruturas internas do software. Essas ferramentas automatizadas possuem um aspecto intrínseco de imprecisão, elas apontam e advertem sobre possíveis defeitos e riscos que possam vir a ocorrer, ou seja, podem ocorrer casos de falso positivo e falso negativo durante as análises. Além disso, a análise estática é considerada um problema computacionalmente indecidível (LANDI, 1992). Isto ocorre porque existem situações no fluxo de execução de uma aplicação analisada que são impossíveis de serem previstas. Apesar dessa limitação, na prática as ferramentas de análise estática produzem resultados úteis (AYEWAH; PUGH, 2008, 2010). Essa imperfeição não as impedem de ter um valor significativo e nem é um fator limitador de seu uso (CHESS; WEST, 2007).

1.1 Motivação

Os mecanismos de tratamento de exceções atuais foram projetados para garantir a robustez dos sistemas (LEE; ANDERSON, 1990). Entretanto, na prática o que se tem observado é dificuldade para se atingir tais objetivos, seja por mau uso dos modelos existentes, seja pela deficiência dos próprios modelos (MILLER; TRIPATHI, 1997; GARCIA et al, 2001; CABRAL; MARQUES, 2007; CACHO *et al*, 2009). Cabral e Marques (2007) realizaram um estudo em que dezesseis aplicações implementadas em C# foram avaliadas. Tais aplicações foram divididas em grupos de quatro e categorizadas da seguinte forma: bibliotecas, aplicações web, servidores e aplicações desktop. Nesse estudo foram identificados alguns exemplos de mau uso dos modelos existentes de tratamento de exceções: (i) lançamento de exceções genéricas que tornam praticamente impossível que se trate e se recupere o erro sem o encerramento do sistema, (ii) captura de exceções genéricas sem o devido tratamento, fazendo com que o software continue a ser executado em estado corrompido, e (iii) ausência de código relativo a tratamento de exceções (lançamento e/ou captura), ocasionando que pequenas situações de erros se tornem grandes problemas. Já como deficiência nos modelos atuais tem-se (CACHO *et al*, 2008): (i) falta de visão global dos fluxos excepcionais, e (ii) suporte limitado para o reuso do comportamento normal e dos manipuladores de exceção.

O problema da falta de visão global é representado pela dificuldade em identificar quem lança e quem trata uma exceção em uma aplicação. A principal conseqüência da falta de visão global do fluxo excepcional é que exceções globais produzem controles de fluxos implícitos (MALAYERI; ALDRICH, 2006), por exemplo, se um programador muda um código relativo a tratamento de exceções, o fluxo de controle em partes aparentemente não relacionadas do programa pode mudar de forma inesperada (ROBILLARD; MURPHY, 2000). Tornando-se difícil descobrir onde as exceções lançadas dentro de uma aplicação serão tratadas e rastrear uma exceção tratada a partir do ponto onde ela foi originalmente lançada.

Além dos problemas citados, estudos de Cabral e Marques (2007) e Ayewah e Pugh (2010) relatam que os desenvolvedores que tentam tratar as exceções de maneira adequada acabam tendo problemas sérios de produtividade. Uma tarefa simples como a leitura de um arquivo em disco pode lançar dez tipos diferentes de exceções. A perda de produtividade eleva os custos, diminui a motivação do desenvolvedor e como consequência diminui a qualidade do software (CABRAL; MARQUES, 2007). Por outro lado, o negligenciamento do

tratamento de exceções, deixando que a linguagem de programação lide com esse problema, embora seja mais produtivo, com código mais limpo e de fácil manutenção, tende a gerar sistemas menos confiáveis e com subseqüente aumento do custo de pós-produção.

O exemplo a seguir ilustra bem como o negligenciamento do tratamento de exceções pode gerar conseqüências graves. Em 1996 o foguete francês Ariane 5 ativou o sistema de autodestruição 37 segundos após decolagem. Após investigação da agência espacial européia descobriu-se que a reutilização de um módulo do Ariane 4 gerou uma exceção durante a tentativa de converter um número de 64 bits para 16 bits. Tal exceção ativou o sistema de autodestruição do foguete (BEN-ARI, 2001).

Outro exemplo de negligência aconteceu no sistema de tratamento e visualização do radar X-4000 utilizado pelo controle de tráfego aéreo brasileiro. Esse sistema é responsável por receber e tratar os dados provenientes dos equipamentos radares, conjugá-los com os dados provenientes dos planos de vôo das aeronaves e gerar a visualização das informações necessárias à prestação do serviço de controle de tráfego aéreo. Após o acidente do vôo GOL 1907 ocorrido em 2006, os controladores de vôo apontaram falhas nesse sistema que contribuíram para o referido acidente. Com base nisso, o Tribunal de Contas da União realizou uma auditoria no sistema X-4000. Em tal auditoria (TCU, 2008) foram observadas falhas de tratamento de exceções em algumas situações em que há ingresso de dados ou comandos não esperados no sistema, comprometendo assim a robustez desse sistema de missão crítica.

Percebe-se com esses dois exemplos que melhorar a robustez de sistemas de software é uma tarefa importante e necessária para a comunidade acadêmica e indústria.

1.2 Objetivos

O presente trabalho baseia-se em pesquisas recentes na área de tratamento de exceções e na plataforma .NET.

Atualmente, o mercado de software para sistemas corporativos e comerciais vem se polarizando basicamente (CABRAL; MARQUES, 2007) entre duas plataformas de desenvolvimento: Java e .NET. Em (CACHO *et al*, 2008, 2009) um estudo abrangente sobre como atingir bons níveis de robustez e manutenibilidade no tratamento de exceções para a plataforma Java foi efetuado. O foco deste trabalho é realizar um estudo similar com a plataforma .NET, tendo como objetivo principal é melhorar a robustez de sistemas de

software compilados para plataforma .NET através de uma ferramenta de analise estática do fluxo excepcional. Como resultado espera-se mais especificamente: (i) o desenvolvimento de uma ferramenta de análise estática focada no comportamento excepcional de aplicações .NET, (ii) utilizar a ferramenta para identificar possíveis defeitos entre diferentes versões de um mesmo sistema de software real. Esta ferramenta deverá permitir que desenvolvedores .NET sejam capazes de:

- Visualizar as métricas coletadas sobre o tratamento de exceções através de tabelas e gráficos.
- Visualizar uma representação gráfica do fluxo excepcional em forma de árvore.
- Acompanhar o comportamento do tratamento de exceções através de várias versões da mesma aplicação.
- Localizar de forma rápida os tipos de exceções lançadas e seus respectivos tratadores.
- Consultar o histórico das análises realizadas.

1.3 Resultados Alcançados

Essa pesquisa de mestrado já produziu alguns resultados. Uma ferramenta de análise estática para a plataforma .NET foi desenvolvida e dois artigos foram publicados.

O primeiro artigo intitulado *eFlowMining: An Exception-Flow Analysis Tool for .NET Applications* foi apresentado no *EhCos - First Workshop on Exception Handling in Contemporary Software Systems*, realizado durante o *Fifth Latin-American Symposium on Dependable Computing Workshops 2011*. Por ter sido aprovado nessa conferência, o artigo também foi publicado no *IEEE Computer Society*. A principal contribuição para essa pesquisa foi a apresentação da primeira versão da ferramenta *eFlowMining* e seu uso em cinco aplicações .NET de diferentes linguagens. É através do uso dessa ferramenta que as hipóteses da pesquisa serão verificadas.

O segundo artigo intitulado *Visualizando a Evolução do Comportamento Excepcional em Aplicações Multi-linguagem com eFlowMining* foi apresentado na *Sessão de Ferramentas* do *II Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2011)* realizado em São Paulo capital. Sendo inclusive premiada como a terceira melhor ferramenta do CBSoft 2011. Nesse artigo uma nova versão da ferramenta *eFlowMining* foi apresentada. Melhorias visuais e no algoritmo de processamento, bem como novas funcionalidades a tornaram a primeira

ferramenta de fluxo excepcional para plataforma .NET a suportar a visualização através de tabelas e gráficos da evolução do comportamento excepcional em várias versões de um mesmo software. Esses dois artigos estão na seção apêndice.

1.4 Organização do trabalho

Esse documento está dividido em cinco capítulos. No Capítulo 2 são apresentados os principais conceitos das áreas de Qualidade de Software, Tratamento de Exceções, Análise Estática, Plataforma .NET e limitações das ferramentas de análise estática do fluxo excepcional. O Capítulo 3 será dedicado a ferramenta *eFlowMining*, com suas funcionalidades, arquitetura e detalhes da implementação. O Capítulo 4 constará da avaliação da ferramenta. Por último, no Capítulo 5 são mostradas as conclusões e trabalhos futuros.

2 FUNDAMENTOS

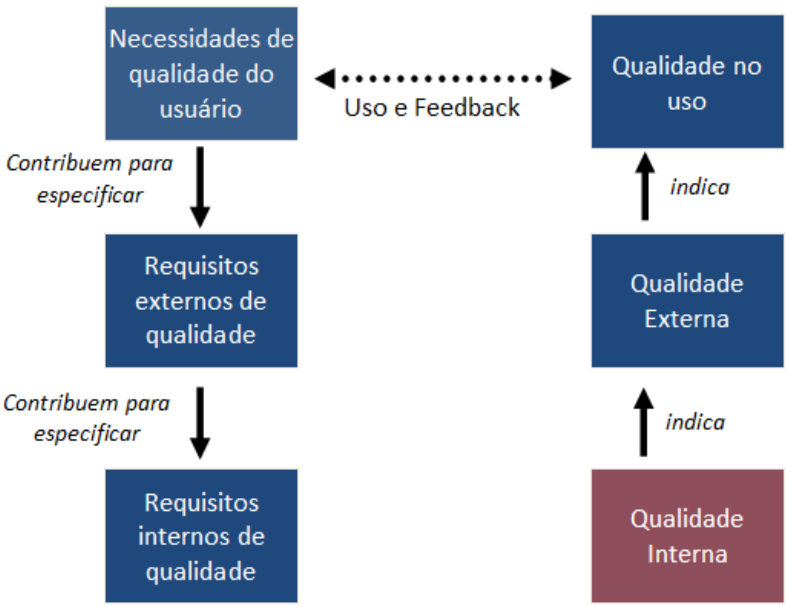
Esse capítulo traz uma discussão dos assuntos teóricos abordados nessa dissertação. São expostos fundamentos e estudos sobre: Qualidade de Software, Análise Estática, Tratamento de Exceções e Plataforma .NET. Na Sessão 2.1 o tratamento de exceções será abordado. Na Seção 2.2 a Plataforma .NET.

2.1 Qualidade de Software

A qualidade dos produtos de software é um fator essencial para o sucesso das empresas que desenvolvem software. Alguns tipos de software, tais como, sistemas de tempo real, software para equipamentos médicos, software controlador de tráfego aéreo, ou qualquer software que lide com aplicações críticas, onde uma falha produza consequências graves, necessitam de atenção especial ao requisito qualidade. Existem várias definições sobre qualidade. Garvin (1984) reuniu várias dessas definições e as categorizou em cinco abordagens de qualidade, a saber: (i) transcendental, pela qual qualidade é algo que se percebe, mas não se define; (ii) do usuário, pela qual qualidade é atingir os objetivos especificados pelo consumidor; (iii) da manufatura, qualidade é seguir precisamente as especificações do projeto; (iv) do produto, vê qualidade como um conjunto mensurável e preciso de características, que são requeridas para satisfazer o consumidor; e (v) do valor, pela qual qualidade está relacionada a quanto o clientes está disposto a pagar pelo produto. Slack, Chambers e Johnston (2002) conciliaram essas abordagens em sua definição de qualidade: *Qualidade é a consistente conformidade com as expectativas dos consumidores.*

Essa definição pode ser relacionada com o modelo de qualidade do padrão ISO/IEC 9126-1:2001 (ISO, 2001) na forma de que na qualidade interna dos produtos, a palavra *conformidade* indica que é preciso seguir processos e especificações durante a fase de produção, e essa qualidade interna irá indicar a qualidade externa, que está relacionada ao produto em execução, que por sua vez levará a qualidade no uso, indicando se as expectativas dos consumidores foram ou não atendidas. A Figura 1 ilustra a relação entre essas visões de qualidade, bem como que cada visão gera necessidades e requisitos específicos que são refinados ao longo do ciclo de vida de produção e uso dos produtos de software.

Figura 1 - Diferentes visões de qualidade. Adaptado (ISO, 2001)



Para cada visão de qualidade descrita na Figura 1, ISO (2001) definiu um conjunto de características e suas respectivas sub-características que podem ser utilizadas para medir o nível de qualidade do software. A Figura 2 ilustra as características das visões interna e externa e a Tabela 1 as descreve.

Figura 2 - Características de qualidade. Adaptado (ISO, 2001)

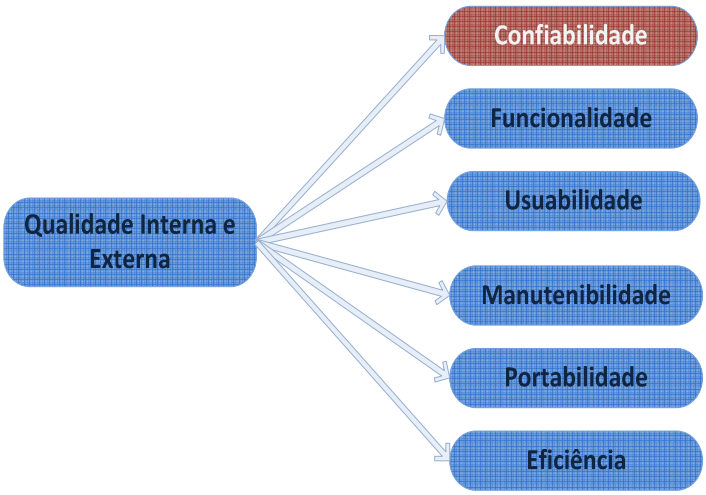


Tabela 1 - Características de qualidade do padrão ISO 9126 (ISO, 2001)

Característica	Descrição
Confiabilidade	Medida da capacidade de um software de manter seu nível de desempenho dentro de condições estabelecidas por um dado período de tempo
Funcionalidade	Conjunto das funções especificadas e suas propriedades. As funções devem satisfazer as necessidades do usuário
Usabilidade	Medida de esforço de uso de um software por um usuário de um determinado perfil
Manutenibilidade	Esforço necessário para fazer alterações no software
Portabilidade	Medida da facilidade do produto de software ser transferido para outro ambiente.
Eficiência	Relação entre o nível de desempenho do software e a quantidade de recursos utilizados.

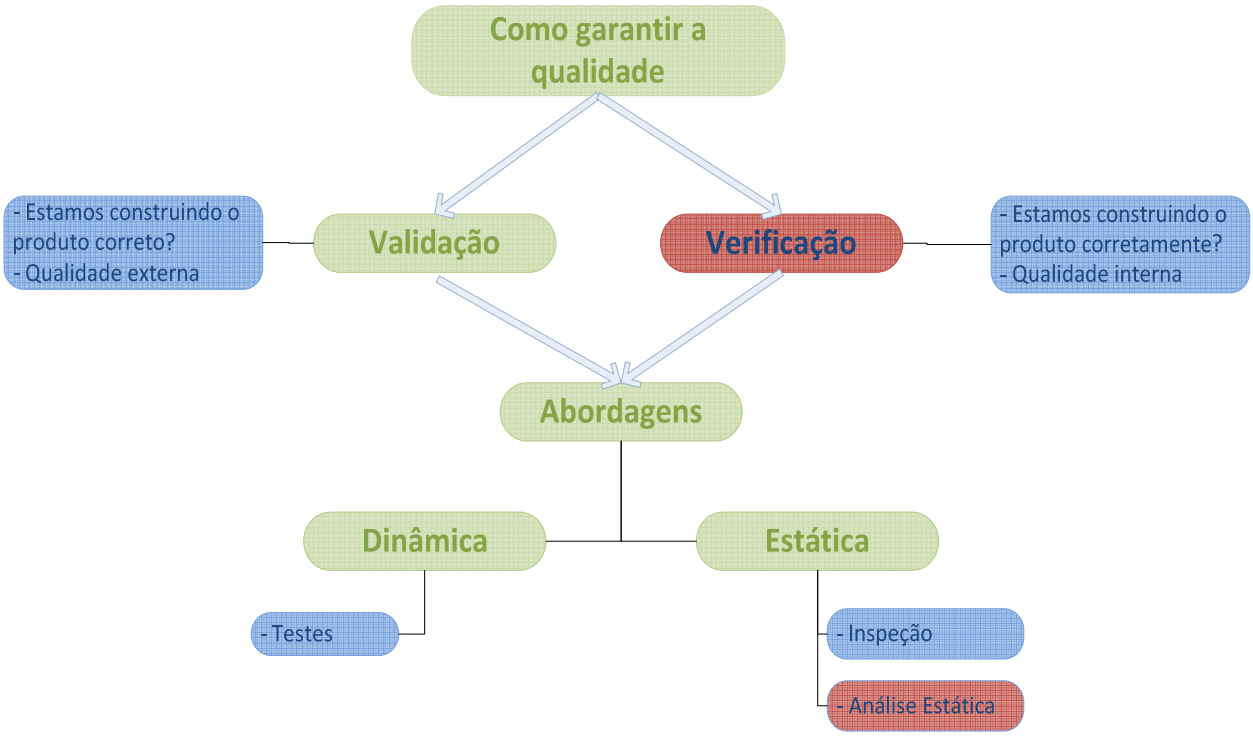
Existem outros modelos de qualidade propostos por acadêmicos e pela indústria. Al-Qutaish (2010) realizou um estudo comparativo entre cinco modelos de qualidade, a saber: McCall's, Boehm's, Dromey's, FURPS e ISO 9126. Embora cada modelo possua nomenclaturas, divisões e enfoques diferentes, o estudo chegou à conclusão que a única característica de qualidade em comum a todos eles foi à confiabilidade, demonstrando a importância desse atributo e a necessidade de processos e ferramentas que assegurem níveis aceitáveis de qualidade.

Atividades de validação e verificação (V&V) são utilizadas para esse fim, elas visam assegurar que tanto o modo como o software está sendo desenvolvido quanto o produto em si estejam em conformidade com a especificação e com as expectativas dos usuários. Boehm (1979) definiu que uma atividade de validação visa garantir que o produto atenda as expectativas dos usuários, já a atividade de verificação visa assegurar que o produto esteja em conformidade com as especificações. Relacionando essas atividades com ISO (2001) conclue-se que a atividade de validação está ligada a qualidade externa do produto, e a atividade de verificação está ligada a qualidade interna.

Essas atividades possuem duas abordagens: (i) estática, realizada através de inspeções manuais nos requisitos, diagramas de projeto e código-fonte, bem como com análises automatizadas do código-fonte; (ii) dinâmica, realizada através de testes de software, envolvendo a execução do software com dados de testes, afim de observar seu comportamento operacional, desempenho e confiabilidade. A abordagem estática pode ser utilizada em qualquer fase do ciclo de vida do software, e serve para verificar a correspondência entre o software e sua especificação, podendo não demonstrar se o software será útil operacionalmente. Para tal, é necessário utilizar a abordagem dinâmica, pois ela é

baseada em cenários de testes e análises de comportamento. A Figura 3 ilustra o relacionamento entre os conceitos explanados acima sobre garantia da qualidade.

Figura 3 - Como garantir a qualidade



2.2 Análise Estática

Revisões e inspeções são tarefas humanas que requerem muita atenção para obterem êxito, bem como alguns cenários de testes podem ser difíceis de serem criados. Nesse contexto, uma técnica bastante utilizada durante a realização de atividades de validação e verificação é a análise estática (AYEWAH; PUGH, 2008, 2010; BESSEY et al, 2010).

Ela consiste de uma análise realizada por ferramentas automatizadas que verificam o software sem executá-lo. Landi (1992) a definiu como o processo de extrair informação sobre a semântica do código em tempo de compilação. Tal extração pode ser feita diretamente no código-fonte ou no código objeto (no caso da plataforma .NET, denominado *ILCode*). A análise direta do código-fonte irá refletir exatamente o que o programador escreveu. Alguns compiladores otimizam o código e outros podem inserir informações ocultas, como o compilador do JavaServer Pages, logo o código objeto poderá não refletir o código-fonte

(CHESS; WEST, 2007). Por outro lado, a análise do código objeto é consideravelmente mais rápida, o que é determinante em projetos com dezenas de milhares de linhas de código.

As ferramentas de análise estáticas podem ser classificadas em quatro grupos de funcionalidades (PFLEEGER; ATLEE, 2009): (i) análise de código, faz verificação de erros de sintaxe, procura por construções com tendência a erros e itens não declarados; (ii) verificação estrutural, mostra relações entre os elementos, aponta possíveis fluxos do programa, detecta loops e partes do código não utilizadas; (iii) análise de dados, verificação da atribuição de valores, revisão das estruturas de dados e verificação da validade das operações; e (iv) verificação de sequencia, verificação da sequencia dos eventos e checagem da ordem das operações.

Além dessa classificação por funcionalidade, as ferramentas podem ser categorizadas de acordo com os problemas que elas se propõem a resolver. A Tabela 2 lista os diversos tipos de ferramentas de acordo com essa classificação (CHESS; WEST, 2007).

Tabela 2 - Tipos de ferramentas de análise estática

Tipo	Descrição
Verificador de tipo	Visa verificar e fazer cumprir as restrições de tipo, por exemplo, associar uma expressão do tipo *int* a uma variável do tipo *short*.
Verificador de estilo	Verifica a conformidade do código-fonte com um estilo de programação pré-determinado. Checando aspectos, tais como, espaço em branco, comentários, nomenclatura das variáveis, funções obsoletas, etc.
Entendimento do programa	Utilizada pelos ambientes integrados de desenvolvimento (*IDEs*) para ajudar o programador a entender o software. Pode ter recursos do tipo: localização de chamadas a um método, localização da declaração de uma variável global, refatoração automática etc.
Verificação do programa	De acordo com uma especificação e um pedaço de código, tenta provar que o código é uma implementação fiel da especificação.
Localização de *bugs*	Aponta locais onde o programa poderá se comportar de uma forma que o programador não tinha a intenção. Tal comportamento é derivado de um conjunto de regras que descrevem padrões de código que geralmente irão resultar em *bugs*.

Essas ferramentas possuem um aspecto intrínseco de imprecisão, elas apontam e advertem sobre possíveis defeitos e riscos que possam vir a ocorrer, ou seja, podem ocorrer casos de falso positivo e falso negativo durante as análises. Falso positivo ocorre quando a ferramenta encontra um problema que na verdade não existe. Já falso negativo ocorre quando o problema existe e não é reportado. Falsos positivos são indesejáveis, pois geram informações irrelevantes, que podem se misturar a problemas reais, porém do ponto de vista da confiabilidade, os falsos negativos são mais preocupantes, pois a partir do resultado gerado pela ferramenta, não há como saber se ocorreu falso negativo ou não, gerando assim uma falsa

sensação de segurança. A análise estática será tão mais precisa quanto menos falsos positivos e falsos negativos produzir, e será confiável se não produzir falsos negativos considerando as propriedades analisadas (CHESS; WEST, 2007).

Além disso, análise estática é um problema computacionalmente indecidível (LANDI, 1992), pois existem situações que são impossíveis de prever quando um algoritmo analisa outro algoritmo, por exemplo, o momento em que o algoritmo que está sendo analisado termina. Esse problema do término de um algoritmo foi identificado por Alan Turing na década de trinta, ele o denominou de *Halting Problem.* Segundo ele, a única forma garantida de saber o que um algoritmo irá fazer é executá-lo, isso significa que se de fato um algoritmo não termina, a decisão sobre o término dele nunca será alcançada (CHESS; WEST, 2007). Além desse problema, em 1953 foi exposto o teorema de *Rice*, que diz que análise estática não pode determinar perfeitamente qualquer propriedade não trivial de um programa.

Apesar desses problemas, na prática as ferramentas de análise estática produzem resultados úteis e ajudam na melhoria da qualidade dos produtos de software (AYEWAH; PUGH, 2008, 2010; BESSEY, 2010). Essa imperfeição não as impedem de ter um valor significativo e nem é um fator limitador de seu uso (CHESS; WEST, 2007). Desta forma, alguns tipos de falhas foram catalogados e indicam formas de utilização comum das ferramentas de análise estática. A Tabela 3 lista alguns desses tipos de falhas.

Tabela 3 - Formas de utilização de ferramentas de análise estática

Tipo	Pode acontecer quando	Como detectar
Race Conditions	Mais de uma *thread* acessa uma mesma variável, sendo pelo menos um dos acessos de escrita e nenhum mecanismo previne esse acesso simultâneo.	Examinar todas as possíveis ordens de comandos das *threads* envolvidas.
Array Bounds	Ocorre acesso à posição inexistente do array, ou seja, quando o índice é menor que a primeira posição ou maior que a última.	Examinar se variáveis inteiras assumem valores menores, iguais ou maiores que o tamanho do array e se são usadas como índice, através da varredura de todos os caminhos possíveis.
Buffer Overflow	Ocorre ao copiar dados para um buffer que possui tamanho menor do que a quantidade de dados de entrada, podendo causar comportamento inesperado no sistema ou furo imperceptível na segurança.	Pode ser detectado de forma semelhante ao *Array Bounds*.
Exception Handling (vide item 2.4)	Não há tratamento de uma exceção lançada.	Examinar exceções lançadas sem bloco `try/catch`.

2.3 Plataforma .NET

Em meados de 2002, a Microsoft lançou oficialmente uma nova iniciativa que repercute no mercado de software mundial. Trata-se de uma plataforma única para desenvolvimento e execução de sistemas, denominada Microsoft.NET, ou como é mais conhecida .NET *Framework.* A idéia principal é que um código gerado para .NET seja executado por qualquer dispositivo que possua o *framework* de tal plataforma, seja ele um computador com Windows ou Linux instalado, um smartphone ou até mesmo um celular. Esse ambiente é composto de ferramentas, *framework* para execução, linguagens de programação e biblioteca de classes que suportam a construção de aplicações desktop, sistemas distribuídos, componentes, páginas dinâmicas para Web e XML Web Services.

Capers Jones (1998) escreveu em seu livro que, pelo menos, um terço das aplicações de software foram implementadas utilizando duas linguagens de programação diferentes. Seguindo essa tendência, a plataforma .NET foi construída para suportar o paradigma multi-linguagem, ou seja, é possível trabalhar com várias linguagens no mesmo projeto e interagir entra elas. Isso ocorre devido ao fato do código ser compilado para uma linguagem intermediaria (*IL Intermediate Language),* e ser somente essa linguagem que o ambiente de execução conhece. A arquitetura dessa plataforma é dividida em camadas, que organiza o caminho que inicia na escrita do código e finaliza com a execução do software. A Figura 4 ilustra esse fluxo e exibe os componentes da arquitetura, a saber:

- **Linguagem de programação**

Não é toda linguagem de programação que pode ser compilada para a linguagem intermediária. Ela dever ser compatível com duas especificações, a saber: (i) *CLS Common Language Specification*, conjunto de recomendações que as linguagens devem seguir e que garante a interoperabilidade entre elas, (ii) *CTS Common Type System,* definição padrão de todos os tipos de dados disponíveis para a *IL*.

- **Metadata (metadados)**

Todo código compilado em .NET é auto-explicativo, ou seja, toda informação necessária para sua execução é armazenada dentro dele próprio na forma de *METATAGS*, fazendo com que o ambiente de execução não precise procurar essas informações no registro do sistema operacional. Algumas dessas informações são listadas a seguir: (i) descrição dos tipos utilizados (classes, estruturas, enumerações, etc), (ii) descrição dos membros

(propriedades, métodos, eventos etc), (iii) descrição das referências a componentes externos (assembly) e (iv) versionamento e integridade do código na seção *manifest*.

- **Assembly**

É a unidade de código física resultante da compilação para *IL*. Pode ser representado por um arquivo executável com extensão .EXE, ou por uma biblioteca de ligação dinâmica com extensão .DLL. Todo *assembly* é auto-explicativo através de seus metadados, conforme explicado no item anterior.

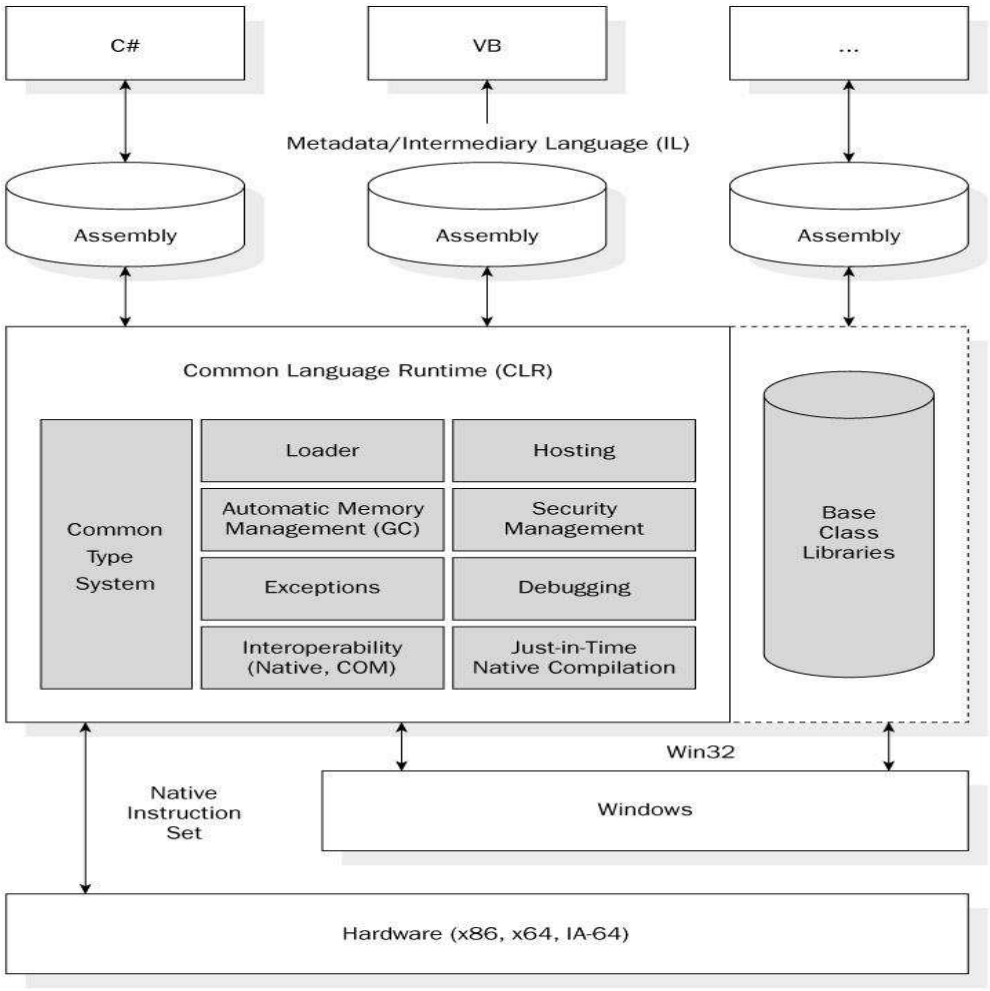
- **CLR – Common Language Runtime**

É o ambiente de execução das aplicações .NET, que funciona como uma máquina virtual que gerencia o relacionamento entre o programa e o sistema operacional. Entre suas responsabilidades estão: gerenciamento de memória, mecanismos de segurança e tratamento de exceções, integração com outras plataformas, por exemplo: COM, depuração, e a compilação *Just-In-Timer (JIT)*. A *JIT* interpreta a *IL* do *assembly* e gera a linguagem de máquina na arquitetura do processador. Existem três tipos de *JIT*: (i) *Pre-JIT*, compila de uma só vez todo o código da aplicação que esta sendo executada e o armazena no cache para uso posterior, (ii) *Econo-JIT*, utilizado em dispositivos móveis onde a memória é um recurso precioso, sendo assim, o código é compilado sob demanda e a memória alocada que não esta em uso é liberada quando o dispositivo assim o requer, (iii) *Normal-JIT*, compila o código sob demanda e joga o código resultante em cache, de forma que esse código não precise ser recompilado quando houver uma nova invocação do mesmo método.

- **Biblioteca de classes (Base Class Library)**

Conjunto de classes para os mais variados propósitos, tais como: acesso a base de dados, gerenciamento de arquivos, gerenciamento de memória, serviços de rede, interface gráfica etc. Este é o principal recurso que possibilita o desenvolvedor criar os sistemas. Sua estrutura é organizada na forma de *namespaces*, baseada em uma hierarquia de nomes. É interessante ressaltar que a *base class library* é desenvolvida para a *IL*, consolidando o aspecto multi-linguagem da plataforma .NET.

Figura 4 - Arquitetura da plataforma .NET



Por ter um código compilado para uma linguagem intermediária, que será interpretado pelo ambiente gerenciado *(CLR)* através do processo *JIT*, a plataforma .NET permite ao desenvolvedor manipular e reescrever *assemblies* dinamicamente e em tempo de execução. Tal recurso é disponibilizado através da funcionalidade *Reflection*, que permite ao desenvolvedor (ROBINSON *et al*, 2004): (i) listar os membros de uma classe, (ii) instanciar um novo objeto, (iii) executar os membros de um objeto, (iv) pesquisar informações sobre uma classe ou assembly, (v) inspecionar os atributos personalizados de uma classe, (vi) criar e compilar um novo assembly e (vii) inserir ou remover instruções *IL* em métodos de uma classe. As principais classes que habilitam o uso do *Reflection* são: `System.Type`, `System.Reflection.Assembly` e `System.Reflection.Emit`.

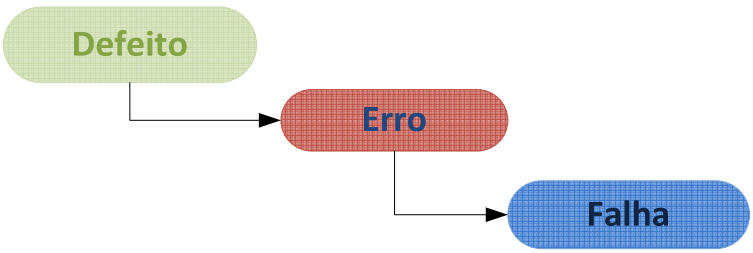
2.4 Tratamento de Exceções

O desenvolvimento de um software robusto requer uma complexidade adicional. Essa complexidade esta relacionada ao código adicional que será utilizado para detectar e tratar defeitos. O mecanismo de tratamento de exceções é a abordagem mais utilizada para resolver os problemas gerados por essa complexidade adicional. Esta seção descreve os conceitos dessa abordagem.

2.4.1 Defeito, Erro e Falha

Um sistema de software consiste de um conjunto de componentes que interagem de acordo com o especificado em um projeto de software, a fim de atender as demandas do ambiente (LEE; ANDERSON, 1990). O projeto define como os componentes irão interagir e estabelece as conexões entre os componentes e o ambiente. Para ser confiável, um sistema de software precisa entregar o serviço esperado mesmo na presença de condições anormais. Para tal, ele recebe requisições de serviços e produz respostas que devem estar de acordo com as especificações ficando assim em um estado consistente. Quando por algum motivo esse estado sofre um desvio, tornando-se inconsistente, é dito que ocorreu um erro (LEE; ANDERSON, 1990; LAPRIE; RANDELL, 2004). Uma falha (LEE; ANDERSON, 1990; LAPRIE; RANDELL, 2004) é a manifestação de um ou mais erros e é de percepção externa, ou seja, pelos usuários do sistema. O problema que pode ter originado o erro, que foi manifestado através de uma falha é denominado defeito, que pode ter origem interna ou externa aos limites do software (LAPRIE; RANDELL, 2004), ou seja, problemas internos no código-fonte ou através de interações errôneas, falhas de hardware etc. A Figura 5 exibe a sequência da ocorrência do defeito até a manifestação da falha.

Figura 5 - Sequência entre defeito, erro e falha.

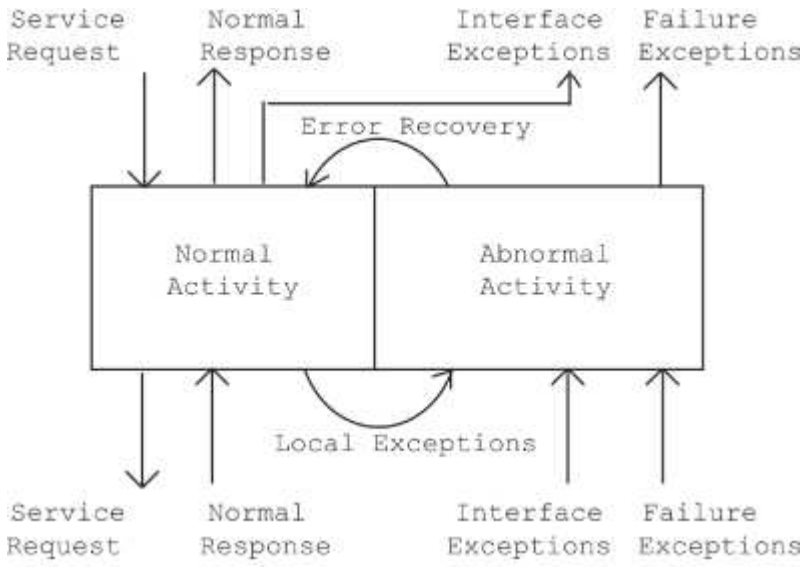


2.4.2 Exceções

Se o sistema não conseguir responder uma requisição de serviço, ele irá retornar uma exceção. Sendo assim, a atividade de um sistema pode ser dividida em: normal e anormal (ou excepcional). A atividade normal corresponde à entrega do serviço que o responsável pela requisição espera, já a atividade anormal provê as medidas necessárias para lidar com as falhas que causaram a exceção. As exceções podem ser divididas em três categorias (LEE; ANDERSON, 1990): (i) exceções de interface, que são sinalizadas em resposta a uma requisição que não está em conformidade com a interface especificada pelo componente; (ii) exceções de defeitos, que são sinalizadas quando o componente determina que por alguma razão não pode prover o serviço requisitado; e (iii) exceções internas, que são exceções geradas pelo próprio componente a fim de invocar suas medidas internas de tolerância a falhas. As exceções são geradas por um componente, porém podem ser sinalizadas entre componentes. Sendo assim, as exceções de interface e de falha podem se tornar exceções externas desde que as ações que irão lidar com ela estiverem fora do componente que as sinalizou. A estrutura e o relacionamento entre essas categorias de exceções foram organizados por Lee e Anderson (1990) em um componente ideal de tolerência a falhas, vide Figura 6.

Ao receber uma requisição de serviço o componente pode responder da seguinte forma: (i) resposta normal, se houver sucesso no processamento; (ii) lançamento de uma exceção de interface, se a assinatura do serviço for inválida; (iii) exceção interna, se ocorrer falha no processamento e o fluxo for desviado para a atividade anormal do componente, e (iv) exceção de defeito, quando a atividade anormal não conseguir voltar o componente para um estado consistente.

Figura 6 - Componente ideal de tolerância a falhas. (LEE; ANDERSON, 1990)



2.4.3 Modelos de Tratamentos de Exceções

A primeira definição de um modelo para o tratamento de exceções foi feita por Goodenough (1975). Este estudo considera que exceções são meios de comunição entre a entidade que invoca a execução de uma operação e algumas condições que podem ocorrem no programa, tais como: erros durante a execução de uma rotina, resultados retornados por uma função, e determinados eventos (não necessariamente falhas) que ocorrem durante a execução do programa.

O gerenciamento das exceções que ocorrem em um sistema é definido como mecanismo de tratamento de exceção (*exception handling*). Tal mecanismo deve contemplar as seguintes atividades: (i) detecção de uma ocorrência de exceção, (ii) desvio do fluxo normal do programa para o fluxo excepcional, (iii) localização do código de tratamento da exceção, e (iv) execução do código que irá lidar com a exceção. Após o tratamento da exceção, a aplicação deve retormar o fluxo normal.

O processo de detecção de uma exceção consiste na sinalização de exceções baseadas em declarações implícitas ou explícitas (GOODENOUGH, 1975). Declarações implícitas são fornecedidas por eventos nativos do hardware que executa o programa e do software, tais como, divisão por zero, ponteiro nulo e overflow. Por outro lado, as declarações explícitas são definidas e sinalizadas pelos próprios desenvolvedores da aplicação. Os mecanismos de tratamento de exceções devem prover construções que permitam tal sinalização. O elemento que detecta o estado anormal e gera a exceção é chamado de sinalizador de exceção

(*exception signaler)*. Após a detecção da exceção, o sistema precisa desviar o fluxo normal do sistema para poder lidar com o problema. O processo interrompe a execução do programa e procura pelo bloco de código associado à situação excepcional. Tal bloco de código é chamado de tratador de exceção *(exception handler)* e é responsável por executar as medidas necessárias para o tratamento do problema. Dependendo do mecanismo utilizado, o tratador de exceção pode ser associado a uma instrução, um bloco, um método, um objeto, uma classe ou uma exceção (GARCIA *et al*, 2001). A região protegida do código aonde o tratador é implementado é chamada de contexto de tratamento de exceções *(EHC)* e é utilizada para limitar o escopo de execução do tratador. Durante o processo de busca do tratador, a exceção pode ser propagada do nível em que se encontra para os níveis mais externos até que um tratador compatível seja localizado.

O controle do fluxo do programa depois que o tratador é executado é determinado pelo modelo de tratamento de exceções. Três modelos são referenciados pela literatura (ROBILLARD; MURPHY, 2003): (i) *termination model*, o escopo que gerou a exceção é destruído, e, se um tratador for encontrado e executado, o fluxo retoma a primeira unidade de código posterior ao tratador; (ii) *resumption model*, após a exceção ser tratada, o fluxo continua do ponto que a exceção foi gerada; (iii) *retry model*, quando a exceção é tratada, o bloco que gerou a exceção é finalizado e então repetido.

2.5 Mecanismos de Tratamento de Exceções na Plataforma .NET

As linguagens de programação que estão em conformidade com as especificações da plataforma .NET suportam os mecanismos de tratamento de exceções de forma nativa. Apesar de possuírem algumas diferenças de sintaxe e semântica, essas linguagens utilizam a mesma essência: (i) uso de blocos `try/catch` para determinar o contexto do tratamento da exceção *(EHC)*; (ii) uso da cláusula `throw` para lançamento da exceção; e (iii) classificação das exceções na forma de uma hierarquia de classes.

Esse suporte, no entanto, não é suficiente para garantir a confiabilidade das aplicações desenvolvidas para a plataforma .NET. É necessário levar em conta o fluxo de controle implícito (MALAYERI; ALDRICH, 2006), pois ele é um fator prejudicial para essa confialiblidade. Fluxos de controle implícitos são tipicamente presentes em aplicações que possuem dependências entre os métodos que são relacionados ao fluxo excepcional (SINHA;

ORSO; HARROLD, 2004). Por exemplo, fluxo de controle implícito gerado pelo fluxo excepcional torna difícil saber: (i) se uma exceção vai ser tratada e em que local, (ii) quais tratadores alternativos estão vinculados a uma exceção específica, (iii) a lista de tratadores destinados para uma exceção, e (iv) que componentes são afetados por um fluxo de controle de exceção.

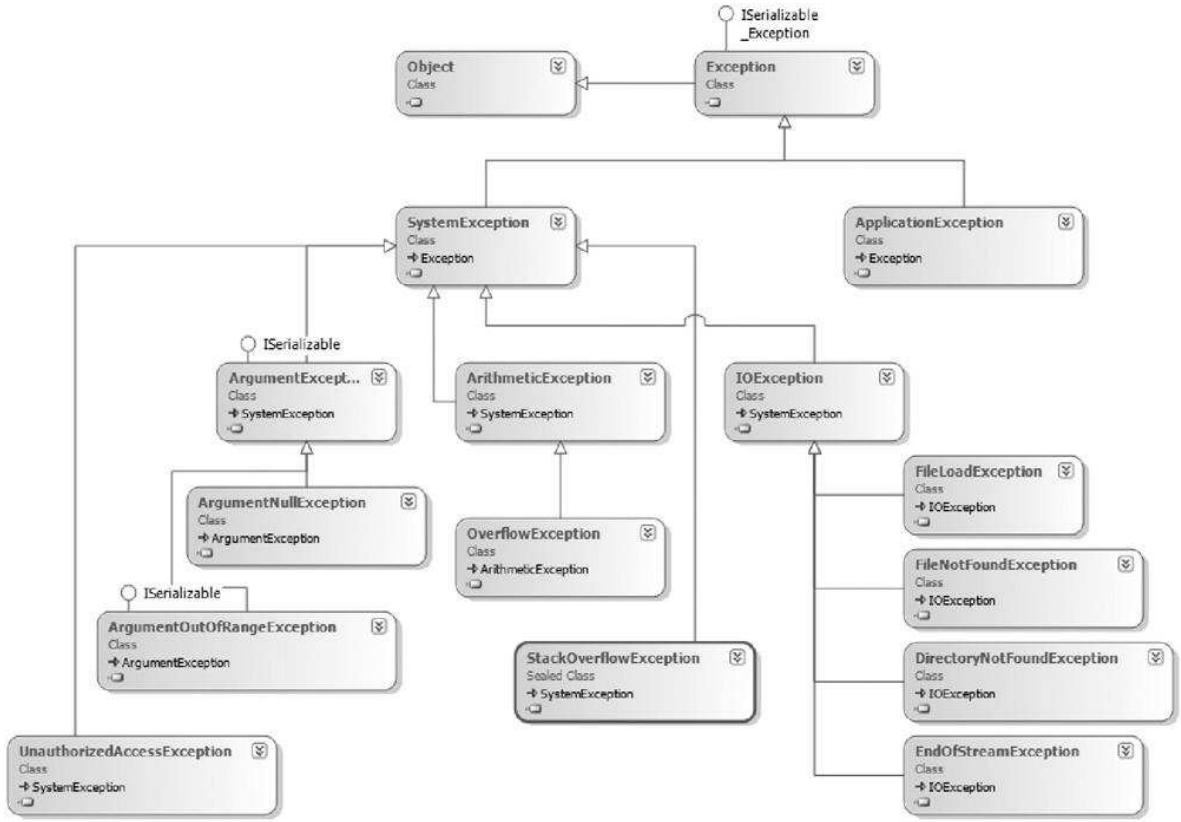
A fim de ilustrar tais cenários nocivos, a seguir será descrito os modelos de tratamento de exceções de cinco linguagens de programação que estão em conformidade com as especificações do .NET *Framework*: C#, F#, J#, VB.NET e C++. Essas linguagens são representativas, pois variam de linguagens orientadas a objetos (C#, VB.NET, J# e C++) a uma linguagem funcional (F#). A hieraquia de classes e alguns aspectos relacionados ao projeto de tratamento de exceções, abordados por Garcia *et al* (2001), serão descritos. Tais aspectos tem um efeito direto na incapacidade dos desenvolvedores em implementar aplicações .NET confiáveis.

2.5.1 Hieraquia de classes

Na plataforma .NET uma exceção é um objeto cujo tipo é uma subclasse de `System.Exception` (RUSTAN; SCHULTE, 2004). Este objeto contém informações que ajudam a rastrear a origem do problema (ROBINSON *et al*, 2004). Embora o desenvolvedor possa criar suas próprias classes de exceção, o .NET Framework possue um número grande de classes pré-definidas para os mais variados tipos de erros. A Figura 7 ilustra a hierarquia de algumas dessas classes. A classe genérica `System.Exception` descende de `System.Object` que é a classe base de todas as classes do .NET. Nessa hierarquia é importante ressaltar as duas classes a seguir:

- `System.SystemException` – Esta classe é para exceções que são lançadas pelo .NET Runtime. Por exemplo, `StackOverflowException` será lançada pelo .NET Runtime na detecção do estouro da pilha. Por outro lado, o desenvolvedor pode lançar a exceção `ArgumentException` ou suas subclasses em seu próprio código, caso se detecte que um método foi chamado com argumentos inválidos;
- `System.ApplicationException` – Esta é a classe base para todas as classes de exceção definidas pelo desenvolvedor, ou seja a definição de exceções próprias do domínio da aplicação deverão descender desta classe.

Figura 7 - Hierarquia de exceções no .NET *Framework* (ROBINSON *et al*, 2004)



A Tabela 4 lista os atributos mais importantes da classe `System.Exception`.

Tabela 4 - Atributos da classe `System.Exception`.

Propriedade	Descrição
`Message`	Texto que descreve a exceção
`HelpLink`	Link para um arquivo de ajuda sobre a exceção
`Source`	Nome da aplicação ou do objeto que causou a exceção
`StackTrace`	Prover detalhes da pilha de chamadas dos métodos, a fim de auxiliar no rastreamento do método que lançou a exceção.
`TargetSite`	Objeto .NET *Reflection* que descreve o método que lançou a exceção
`InnerException`	Objeto `Sytem.Exception` que causou a exceção

Essa representação permite a ocorrência de *subsumption* (ROBILLARD; MURPHY, 2003): quando um objeto de um subtipo pode ser atribuído a uma variável declarada como sendo de seu supertipo. Nesse caso, quando uma exceção é sinalizada e não existe um tratador especifico, ela pode ser capturada por um tratador que trata um supertipo da mesma.

2.5.2 Interface de Exceção

A interface de exceção é utilizada em algumas linguagens de programação para explicitar que exceções podem ser lançadas por um método (LANG; STEWART, 1998). Tal recurso é útil na medida em que o chamador de um método pode preparar seu código para as condições excepcionais que podem ocorrer durante a execução do programa. Sendo assim, no âmbito das linguagens de programação as exceções são divididas em: (i) exceções checadas, que precisa ser declarada na assinatura do método; e (ii) exceções não-checadas, que não precisa ser declarada.

A fim de controlar os fluxos de controle implícito, interface de exceção deveria ser obrigatória. Todas as exceções propagadas por um método deveriam ser rigorosamente especificadas na assinatura do mesmo. Interface de exceção não é suportada em C#, F# e VB.NET. Para essas linguagens, os desenvolvedores precisam examinar a implementação e documentação do método para identificar que exceções podem ser propagadas. Cabral, Sacramento e Marques (2007) fizeram uma extensa análise sobre o uso da documentação das exceções em aplicações .NET. Os autores concluíram que 87% das exceções lançadas não foram documentadas. Sendo assim, nas linguagens citadas acima, a documentação sobre as exceções provê um suporte limitado que ajuda os desenvolvedores a determinar se uma chamada a um método está lançando uma exceção.

Para evitar essa situação, até certo ponto, C++ adota a abordagem opcional para interface de exceção. Essa abordagem oferece a palavra reservada `throw` para definir interfaces de exceção para os métodos. Métodos com a cláusula `throw` só podem propagar exceções listadas na interface. Entretanto, se nenhuma exceção for listada na assinatura do método, ele poderá lançar qualquer exceção.

Uma abordagem híbrida é suportada pelo J# através do suporte a exceções checadas e não-checadas. O compilador força as exceções checadas a serem associadas a um tratador ou definidas explicitamente na interface de exceção. Por outro lado, exceções não-checadas não obrigam que o programador nem associe a um tratador nem especifique a interface de exceção. Para separar exceções entre checadas e não-checadas, o J# define uma hierarquia de tipos de exceções que caracteriza três grupos semânticos e funcionais: *errors*, exceções de *runtime* e exceções *checked*. *Errors* e exceções de *runtime* são não-checadas pelo compilador e não precisam ser declaradas na interface de exceção.

Ainda que o mecanismo de interface de exceção fornecido pelo J# ajude a encontrar fluxos de controle implícito, ele ainda não torna possível fazer uma ligação precisa e explicita do local de lançamento da exceção com o tratador desejado. Ele só indica o potencial caminho de propagação da exceção. Não tem como garantir que as exceções sejam sempre capturadas pelos tratadores corretos.

2.5.3 Vinculação de Tratadores

Contexto de tratamento de exceções *(Exception handling contexts - EHC)* são regiões de código do programa onde os tratadores são vinculados. Um EHC pode ter um conjunto de tratadores associados, entre os quais um é escolhido em tempo de exceção quando uma exceção é lançada dentro do contexto. Todas as linguagens analisadas suportam somente o EHC do tipo *block*. Um bloco é definido por um construtor especial chamado *try* que delimita o conjunto de instruções associadas ao tratador. O bloco *try* inclue uma instrução composta, chamada de cláusula `try`, e uma lista de tratadores de exceção associados. A Tabela 5 descreve como os tratadores de exceção são especificados em cada uma das linguagens analisadas.

Tabela 5 - Vinculação de tratadores de diferentes linguagens de programação

Linguagem	Bloco EHC
J#	try {S} catch (E1 x) {T} catch (E2 x) {T}
C#	try {S} catch (E1 x) {T} catch (E2 x) {T}
C++	try {S} catch (E1 x) {T} catch (E2 x) {T}
VB.NET	try {S} Catch x As E1[When c] T Catch x As E2 T End Try
F#	try S With | :? E1 -> T | :? E2 -> T | :? C -> T

As políticas de vinculação de tratador podem apresentar um impacto negativo no fluxo de controle excepcional. Isso decorre do fato que escopos de tratamento maiores, como blocos, não deixam explícito que instrução dentro do bloco *try* está atualmente sinalizando uma exceção. Como esta informação não é facilmente obtida devido à falta de documentação (CABRAL; SACRAMENTO; MARQUES, 2006, 2007), desenvolvedores não podem determinar se o tratador de exceções está de acordo com todas as instruções do bloco *try* (ROBILLARD; MURPHY, 2003). Como apontado por muitos autores (REIMER;

SRINIVASAN, 2003; ROBILLARD; MURPHY, 2003; SINHA; ORSO; HARROLD, 2004), essa definição falha sobre os locais de lançamento de exceções pode, inconscientemente, levar a duas indesejáveis, porém comuns, situações: um único tratador para múltiplas exceções não relacionadas (REIMER; SRINIVASAN, 2003) e tratadores inacessíveis (ROBILLARD; MURPHY, 2003; SINHA; ORSO; HARROLD, 2004).

Na primeira situação, um programador pode inserir uma nova instrução que lança exceção dentro de um EHC onde um tratador genérico já está vinculado. A intenção dele é que a nova exceção seja capturada por um tratador específico. Uma vez que o compilador não reclama, o programador não altera o tratador, e como resultado disso, a exceção pode eventualmente ser tratada de forma inadequada pelo tratador genérico associado ao EHC. A segunda situação geralmente acontece quando o programador remove instruções de um tratador que já foi vinculado. Nesse caso, por prudência (ROBILLARD; MURPHY, 2003), o tratador não foi removido e se transformou em um código morto. Da mesma forma que a primeira situação, o problema surge quando a instrução nova, que sinaliza tipos de exceções similares para a instrução removida, é inserida dentro do EHC para o tratador morto. Novamente, o tratador pode ser levado a tratar uma exceção que não foi projetado para tal.

2.5.4 Ligação de Tratador

Ligação de tratador determina como o mecanismo de tratamento de exceções procura o tratador. Todas as linguagens analisadas neste artigo suportam a abordagem *semi-dynamic* (GARCIA *et al*, 2001). Nessa abordagem o mecanismo de tratamento de exceções executa uma procura por tratadores levando em conta os tratadores vinculados estaticamente ao EHC. Na sintaxe descrita na Tabela 5, o código em *S* pode gerar uma exceção. Quando uma exceção é lançada, cada padrão *x* é comparado com o tipo de exceção *En*, e para a primeira comparação bem sucedida o tratador respectivo é executado. Se nenhum padrão for encontrado, a exceção será propagada através da pilha de chamadas até um tratador ser localizado. Se nenhum tratador for localizado, o mecanismo de tratamento de exceções propaga uma exceção genérica ou finaliza o programa.

O uso de pesquisa de tratador de forma semi-dinâmica contribui para que o fluxo de controle implícito seja mais difícil de ser entendido. A ligação de tratador semi-dinâmica confia na pesquisa através da pilha de chamadas para localizar um tratador para uma exceção. Como a pilha de chamadas só esta disponível em tempo de execução, os desenvolvedores

raramente sabem precisamente onde a exceção será tratada (YEMINI; BERRY, 1985; CACHO *et al*, 2008) dado que o tratador possa está a um, dois ou três níveis acima na pilha de chamadas.

Outro problema pode ocorrer quando o mecanismo de ligação do tratador manipula o tipo ou alguma informação do contexto da aplicação para localizar um tratador. Por exemplo, todas as linguagens analisadas permitem que as exceções sejam comparadas por *subsumption*. Uma exceção é capturada por subsunção quando o tipo associado ao tratador é supertipo do tipo de exceção lançada. Adicionalmente, linguagens como F# e VB.NET suportam filtros de exceção que possibilita ao desenvolvedor construir uma cláusula `catch` condicional. A Figura 8 ilustra a utilização de um filtro de exceção através do construtor `When` em VB.NET. Nesta figura, o tratador somente será executado quando a exceção *e* for um subtipo de `IOException` e se a condição (*filename* <> *String.Empty*) for verdadeira. Se a condição for falsa, o sistema irá continuar a procurar por tratadores através da pilha de chamadas.

Como dito por vários autores (MILLER; TRIPATHI, 1997; ROBILLARD; MURPHY, 2000, 2003) essas duas abordagens são particularmente prejudiciais quando o programa evolui. Miller e Tripathi (1997) expõem muitas situações onde a evolução do programa compromete a eficácia do comportamento do tratamento de exceções. Todas essas situações são baseadas no fato que um módulo pode ser alterado para poder lançar exceções adicionais, enquanto que módulos inalterados têm que lidar com elas usando os tratadores já existentes. Como conseqüência disso, tratamento de exceções por subsunção torna praticamente impossível assegurar que as exceções serão tratadas especificamente dado que novas exceções podem ser incluídas, e assim sobrecarregar o tipo das já existentes.

Figura 8 - Exemplo de uso de filtro em VB.NET

```
Dim namefile As String = "c:\ex.txt"

 Try
   Dim fs As New IO.FileStream(namefile,FileMode.CreateNew)
 Catch e As IOException
    When namefile <> String.Empty
      Console.Write("The file already exists!" & vbCrLf)
      Throw New IOException("That file already exists on the hard
drive!" & vbCrLf)
 Catch e As Exception
      Console.Write("The message was : " & e.Message & vbCrLf)
      Console.Write("The stacktrace was : " & e.StackTrace & vbCrLf)
  End Try
```

2.6 Limitações das ferramentas de análise estática do fluxo excepcional

O uso de ferramentas de análise estática é uma abordagem clássica para mitigar problemas criados por fluxos excepcionais implícitos. Existem algumas ferramentas (SCHAEFER; BUNDY, 1993; FAHNDRICH *et al*, 1998; CHANG *et al*., 2001; ROBILLARD; MURPHY, 2003; CABRAL; MARQUES; SILVA, 2005; FU; RYDER, 2007; SHAH; CARSTEN; HARROLD, 2008; COELHO *et al*, 2008) que, baseadas no gráfico de chamadas de um programa, rastreiam o fluxo das exceções através dos caminhos desse gráfico, além de coletar métricas sobre o comportamento excepcional. Elas fornecem uma ajuda valiosa para os desenvolvedores tentarem entender como o programa se comporta na presença de exceções globais, e como as exceções se comportam.

Geralmente, essa informação consiste de um caminho de propagação das exceções e é usada, por exemplo, para identificar exceções não tratadas em linguagens com tipos polimórficos. A seguir algumas ferramentas encontradas:

- Chang *et al* (2001) apresentou uma ferramenta de análise estática para programas Java que estimava seus fluxos excepcionais. Essa análise foi utilizada para detectar especificações de exceções genéricas ou desnecessárias e tratadores.
- Fahndrich *et al* (1998) utilizaram o toolkit BANE para descobrir exceções não tratadas na linguagem funcional ML.
- O trabalho de Schaefer e Bundy (1993) descreve um modelo para entender os fluxos excepcionais em programas Ada. Eles também apresentam uma ferramenta que usa esse modelo para rastrear exceções não tratadas e provê informações sobre o fluxo excepcional para os programadores.
- A popular ferramenta JEX, planejada por Robillard e Murphy (2003), analisa o fluxo excepcional em programas Java. Ela inclui uma GUI que mostra o caminho de propagação das exceções e detecta tratadores genéricos.
- Fu e Ryder (2007) propuseram uma extensão para a típica análise estática em Java. Em vez de tentar identificar os caminhos de propagação de exceções, eles analisam o destino que eles chamam de cadeias de propagação de exceções. Eles tentam descobrir relações casuais entre os caminhos de propagação de exceções a fim de provê uma imagem do sistema inteiro, em temos de como os fluxos excepcionais são inter-relacionados.

- Shah *et al* (2008) descreveu a ferramenta EnHanCe que suporta uma grande variedade de informações sobre construtores de tratamento de exceções e fluxos excepcionais através de perspectivas quantitativas, fluxo e contextual. Ela foi implementada em Java e pode ser utilizada como um plugin no IDE Eclipse.
- Cabral *et al* (2005) desenvolveu a ferramenta *RAIL - Runtime Assembly Instrumentation Library*. Focada na plataforma .NET, ela foi utilizada para extrair informações sobre o comportamento excepcional de programas .NET, tais como métodos que lançam exceções, tratadores e EHCs, em um estudo realizado por Cabral e Marques (2007) com o intuito de avaliar como os programadores utilizam os mecanismos de tratamento de exceções. A última versão (v0.5.7) dessa ferramenta foi lançada em janeiro de 2005.
- Coelho *et al* (2008) desenvolveu uma ferramenta de análise estática do fluxo excepcional com propósito de avaliar o impacto da programação orientada a aspectos nos fluxos excepcionais de programas desenvolvidos em AspectJ. Ela foi desenvolvida baseada no *framework* Soot para análise do código objeto, além do módulo Spark do próprio Soot para construção do gráfico de chamadas.

A Tabela 6 consolida as seguintes características das ferramentas listadas acima: (i) exibe fluxos graficamente, se a ferramenta permite a visualização dos fluxos excepcionais na forma de árvore ou de partes do gráfico de chamadas; (ii) registra histórico das análises, se as análises são armazenadas em banco de dados, arquivos XML, etc; (iii) acompanha evolução do fluxo excepcional, se a ferramenta compara o fluxo excepcional de versões diferentes do mesmo programa e exibe as diferenças; (iv) alvo da análise, se a análise é feita no código fonte ou código objeto do programa.

Tabela 6 - Características das ferramentas de análise estática do fluxo excepcional

Autor	Nome ferramenta	Linguagem alvo	Exibe fluxos graficamente	Registra histórico das análises	Acompanha evolução do fluxo excepcional	Alvo da análise
Chang *et al* (2001)	Não possui	Java	Não	Não	Não	Código Fonte
Fahndrich *et al* (1998)	Não possui	ML	Não	Não	Não	Código Fonte
Schaefer e Bundy (1993)	Não possui	Ada	Não	Não	Não	Código Objeto
Robillard e Murphy (2003)	JEX	Java	Sim	Não	Não	Código Fonte
Fu e Ryder (2007)	Não possui	Java	Não	Não	Não	Código Objeto
Shah *et al* (2008)	Enhance	Java	Sim	Não	Não	Código fonte
Cabral *et al* (2005)	RAIL	.NET	Não	Não	Não	Código Objeto
Coelho *et al* (2008)	SAFE	Java/AspectJ	Sim	Não	Não	Código Objeto

Analisando a Tabela 6 percebe-se que a maioria das ferramentas são focadas na plataforma Java, somente o estudo (CABRAL; MARQUES; SILVA, 2005) examinou a plataforma .NET, ele porém não implementou a visualização gráfica dos fluxos excepcionais em forma de árvore. Essa visualização também não foi trabalhada na maioria das ferramentas, somente por (ROBILLARD; MURPHY, 2003) e (SHAH; CARSTEN; HARROLD, 2008). Além disso, em nenhuma ferramenta foi implementado o registro histórico das análises e o acompanhamento da evolução do fluxo excepcional entre as versões das aplicações. Esses dois requisitos, bem como a visualização gráfica dos fluxos excepcionais, são aspectos que ajudariam a aumentar a objetividade das análises. O registro histórico permitiria que uma análise fosse armazenada em algum mecanismo de persistência: banco de dados, arquivo XML por exemplo. Tal registro poderia servir para reanalises futuras ou servir de base para o requisito de acompanhamento da evolução dos fluxos, pois a partir das análises armazenadas seria possível comparar os fluxos que surgirem/desaparecem entre essas análises.

3 EFLOWMINING: UMA FERRAMENTA DE ANÁLISE DE COMPORTAMENTO EXCEPCIONAL PARA APLICAÇÕES .NET

A análise do comportamento excepcional de sistemas desenvolvidos para a plataforma .NET, independente da linguagem utilizada: C#, VB.NET, J#, etc, é uma tarefa importante para a medição do nível de robustez de sistemas de software. Além disso, essa análise pode ajudar os programadores a mitigarem os problemas decorrentes do controle de fluxo excepcional implícito descrito na seção 2. Nesse contexto foi desenvolvida a ferramenta de análise estática do fluxo excepcional *eFlowMining*, cujo objetivo é fornecer aos programadores .NET uma forma de analisar os fluxos excepcionais de aplicações compiladas para plataforma .NET. Tal análise pode ser feita a partir das diversas visões providas pela ferramenta que serão apresentadas nas seções seguintes. De acordo com as categorizações das ferramentas de análise estática realizadas na seção 2.2, a *eFlowMining* pode ser enquadrada da seguinte forma: categoria verificação estrutural, pois procura por fluxos excepcionais; classificação localizador de *bugs*, pois irá apontar exceções não tratadas; falha conhecida tratamento de exceções, pois irá focar nesse problema.

3.1 Funcionalidade

As funcionalidades da *eFlowMining* foram definidas a partir das limitações das ferramentas de análise estática do fluxo excepcional pesquisadas no estado da arte. Além disso, também foi levado em conta aspectos práticos vivenciados pelos desenvolvedores .NET, tais como: necessidade de armanezamento das informações coletadas em banco de dados e interface gráfica amigável para visualização.

Para o devido entendimento das funcionalidades que serão descritas nessa seção é importante relembrar as seguintes características da plataforma .NET já vistas na seção 2: (i) o código .NET é compilado para uma linguagem intermediária (*IL – Intermediate Language*), sendo somente essa linguagem que o ambiente de execução processa; (ii) *assembly* é a unidade de código física resultante da compilação para IL, podendo ser representado por um arquivo executável com extensão .EXE, ou por uma biblioteca de ligação dinâmica com externsão .DLL.

A seguir os requisitos funcionais são listados:

- A ferramenta deve permitir a escolha de *assembly* .NET no sistema operacional

para coleta das métricas.

- A ferramenta deve recuperar o número da versão do *assembly* selecionado.
- A ferramenta deve permitir que se informe a linguagem de programação e um nome amigável para o *assembly* selecionado.
- A ferramenta deve processar o *assembly* selecionado gravando em banco de dados o gráfico de chamadas, as métricas listadas na Tabela 7 e os fluxos excepcionais.
- A ferramenta deve permitir visualizar o resultado do processamento das várias versões do mesmo arquivo, através das visões listadas na Tabela 8.
- A ferramenta deve permitir a localização rápida dos tipos de exceções lançadas e seus respectivos tratadores.

Tabela 7 - Métricas providas pela *eFlowMining*

Métricas	Descrição
Types	Quantidade de classes
Methods	Quantidade de métodos
Try	Quantidade de classes de blocos *try*
Try Size	Média da quantidade de instruções IL dentro de um bloco *try*
Handler	Quantidade de tratadores
Handler Size	Média da quantidade de instruções *IL* dentro de blocos tratadores
Generic Handler	Quantidade de tratadores vinculados a supertipos de exceções, por exemplo, `System. Exception`.
Specialized Handler	Quantidade de tratadores vinculados a exceções específicas.
Filter Handler	Quantidade de tratadores com condições de filtro.
Application Throw	Quantidade de fluxos excepcionais que estão sendo gerados dentro do *assembly* que está sendo analisado.
Reference Throw	Quantidade de fluxos excepcionais que estão sendo gerados em *assemblies* referenciados ou dentro do .NET *Framework*.
Specialized Exception Flow	Quantidade de fluxos excepcionais em que a exceção lançada é do mesmo tipo da exceção capturada.
Subsumption Exception Flow	Quantidade de fluxos excepcionais em que a exceção lançada é subtipo da exceção capturada.
Uncaught Exception Flow	Quantidade de fluxos excepcionais em que a exceção lançada não foi capturada.
Call Stack Levels	Média da quantidade de níveis da pilha de chamadas por onde a exceção viaja antes de ser capturada.

Essas métricas podem ajudar os desenvolvedores a melhorar robustez de aplicações .NET de várias formas:

- As métricas *Generic Handler*, *Filter Handler* and *Subsumption Exception Flow* podem ser usadas para identificar a ocorrência de exceções capturadas por subsunção (ROBILLARD; MURPHY, 2003) ou quando filtros forem aplicados.

- A métrica *Try Size* pode indicar uso inadequado de blocos *try* grande. Um alto escopo de tratamento pode ser a causa de alto número de *Uncaught Exception Flow* para essa aplicação.
- A métrica *Handler Size* suporta a identificação de exceções desprezadas. Por exemplo, uma média baixa de instruções *IL* dentro de um tratador pode revelar que alguns tratadores estão vazios ou não estão tratando as exceções.
- A métrica *Reference Throw* expõem a quantidade de exceções lançadas por bibliotecas de terceiros que são geralmente escondidas por interfaces de exceções imprecisas ou documentação falha.
- A métrica *Call Stack Levels* pode revelar alguns desvios na ligação de tratador semi-dinâmica. Uma alta média na quantidade de *Call Stack Levels* pode indicar que o tratamento de exceções está pouco significativo ou com dificuldade de depuração (REIMER; SRINIVASAN, 2003; SINHA; ORSO; HARROLD, 2004).
- A métrica *Uncaught Exception Flow* pode indica a presença de possíveis falhas uma vez que exceções criadas pela aplicação ou aplicação de terceiros não estão sendo tratadas.

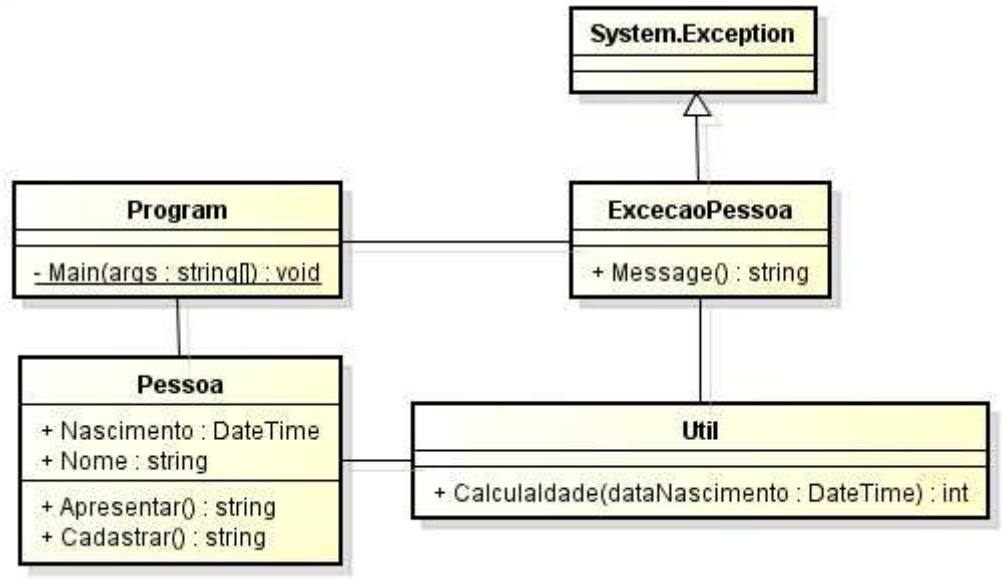
Tabela 8 - Visões providas pela *eFlowMining*

Visão	Descrição
Metric	Exibe as métricas coletadas para as diversas versões do *assembly* selecionado.
Reference	Exibe os assemblies externos referenciados, bem como as referências ao .NET *framework*.
Types	Exibe as classes e seus métodos.
Exception Types	Exibe um resumo com a quantidade dos diversos tipos de exceções lançadas e capturadas.
Exception Flow	Exibe graficamente os fluxos excepcionais na forma de árvore.
Evolution	Exibe um detalhamento do comportamento do tratamento de exceções através das versões da aplicação.
Graph	Exibe um gráfico de linha com a evolução de uma ou mais métrica através das versões da aplicação.

Para explicar as visões da Tabela 8 com dados reais, um programa exemplo foi desenvolvido em C#. Foram escritas três versões, a primeira sem nenhum tratamento excepcional, a segunda com tratamento genérico e a terceira com tratamento especializado, além disso, alguns fluxos novos foram incluídos entre elas. Sua única funcionalidade é a exibição dos dados instanciados em uma classe, porém, apesar dessa simplicidade poderemos

visualizar os fluxos excepcionais e verificar os cenários falhos entre as versões. O diagrama de classes do programa é exibido na Figura 9.

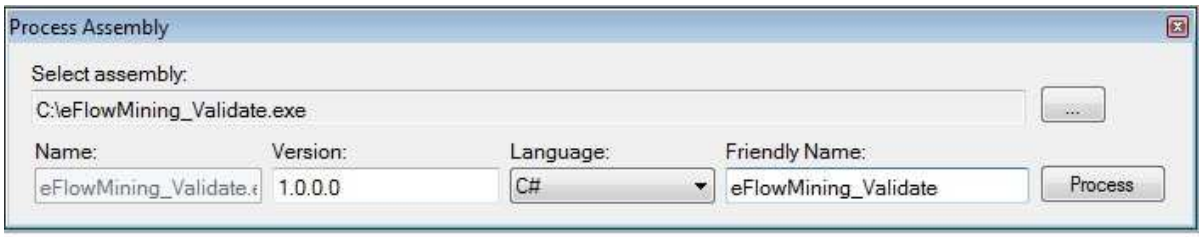
Figura 9 - Diagrama de classes do programa exemplo



A classe *Pessoa* possui os atributos *Nome* e *Nascimento*, e os métodos *Apresentar* e *Cadastrar* que acessam o método *CalculaIdade* da classe *Util*. A classe *ExcecaoPessoa* é subclasse da classe genérica de exceção do framework .NET *System.Exception* e sobrescreve a propriedade *Message* (vide seção 2.5.1). A classe *Program* é utilizada para iniciar o programa.

O primeiro passo no uso da ferramenta é o processamento do *assembly*. Ao selecionar o arquivo, as informações da versão são exibidas, devendo ser informado um nome amigável e em qual linguagem foi desenvolvido. É possível processar várias versões do mesmo *assembly*. A Figura 10 exibe essa tela.

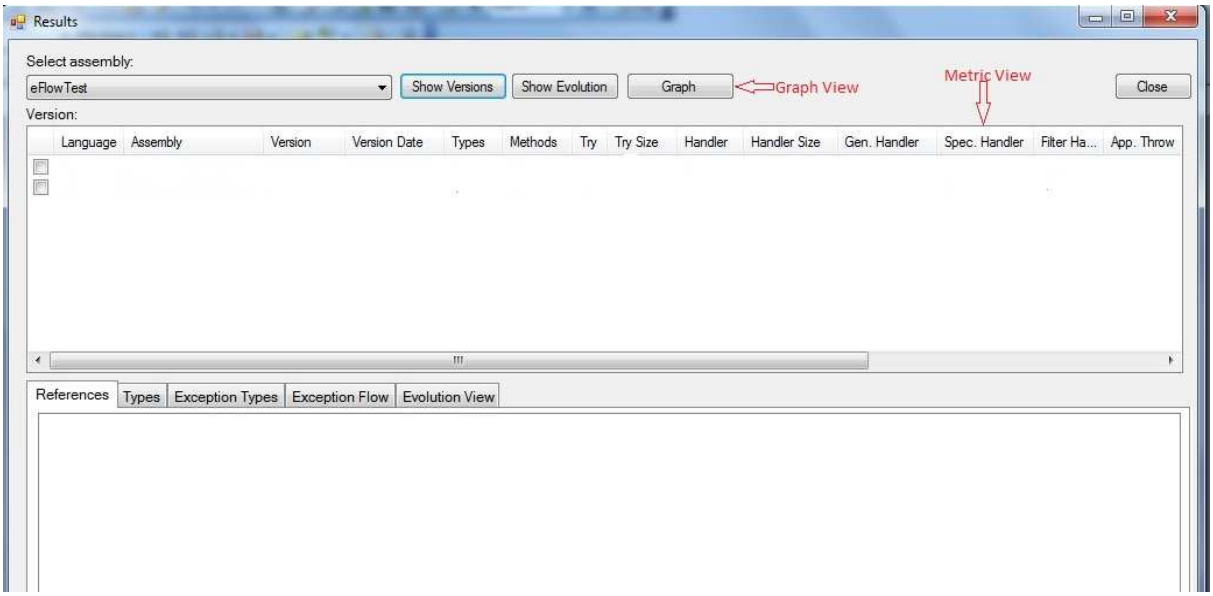
Figura 10 - Tela de processamento de *assembly* da *eFlowMining*



O processamento irá popular o banco de dados com as métricas e os fluxos excepcionais de acordo com o algoritmo que será descrito na próxima seção. Após o processamento, o programador poderá acessar a tela de resultados a fim de realizar consultas às métricas e

visões disponíveis (vide Tabela 7 e Tabela 8, respectivamente). A tela principal de resultados é exibida na Figura 11.

Figura 11 - Tela de resultados da *eFlowMining*

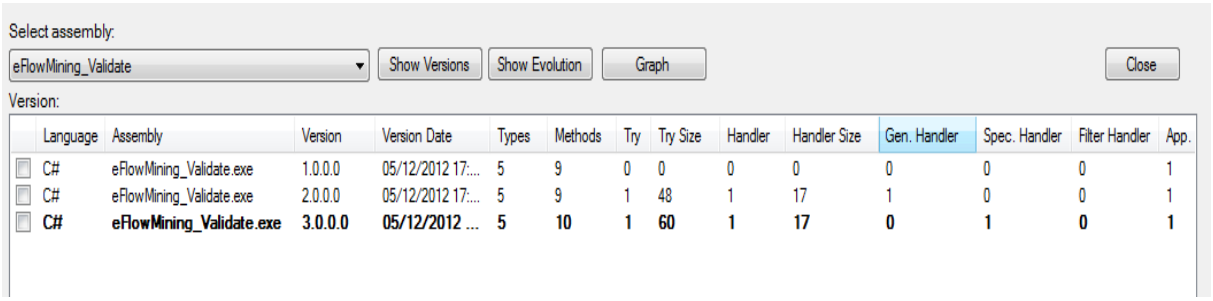


As setas em vermelho na parte superior da Figura 11 apontam para a visão *Metric* e para o botão que dá acesso a visão *Graph*. As visões *Reference, Types, Exception Types, Exception Flow e Evolution* estão dispostas em abas na parte inferior da tela.

A Figura 12 exibe a visão *Metric* para três versões do programa exemplo. Cada linha da tabela representa uma versão da aplicação selecionada. A *Metric View* ajuda os desenvolvedores .NET a ter uma visão geral sobre como a aplicação esta implementada no que diz respeito aos mecanismos de tratamento de exceções.

Através do agrupamento de diferentes versões da mesma aplicação em uma única visão, a *Metric View* também provê ao desenvolvedor a visualização do comportamento das várias métricas durante a evolução da aplicação. Isso permite identificar as mudanças de cenários que tem efeitos positivos ou negativos para o comportamento do tratamento de exceções.

Figura 12 - Visão *Metric*



Uma visão detalhada do comportamento do tratamento de exceções pode ser visualizada na *Evolution View* (Figura 13). Essa visão ajuda a identificação de cenários problemáticos (na forma de X/Y) mostrando para cada método e versão da aplicação o número de exceções lançadas (X) e o número de fluxos que foram tratados (Y). A ferramenta também muda a cor da linha quando houver diminuição de exceções tratadas entre uma versão e outra, e quando houver aumento de exceções lançadas. Por exemplo, a ferramenta indica que entre a versão 2.0.0.0 e a 3.0.0.0 do método `Pessoa.Cadastrar()` foram lançadas três exceções e nenhuma delas foram tratadas.

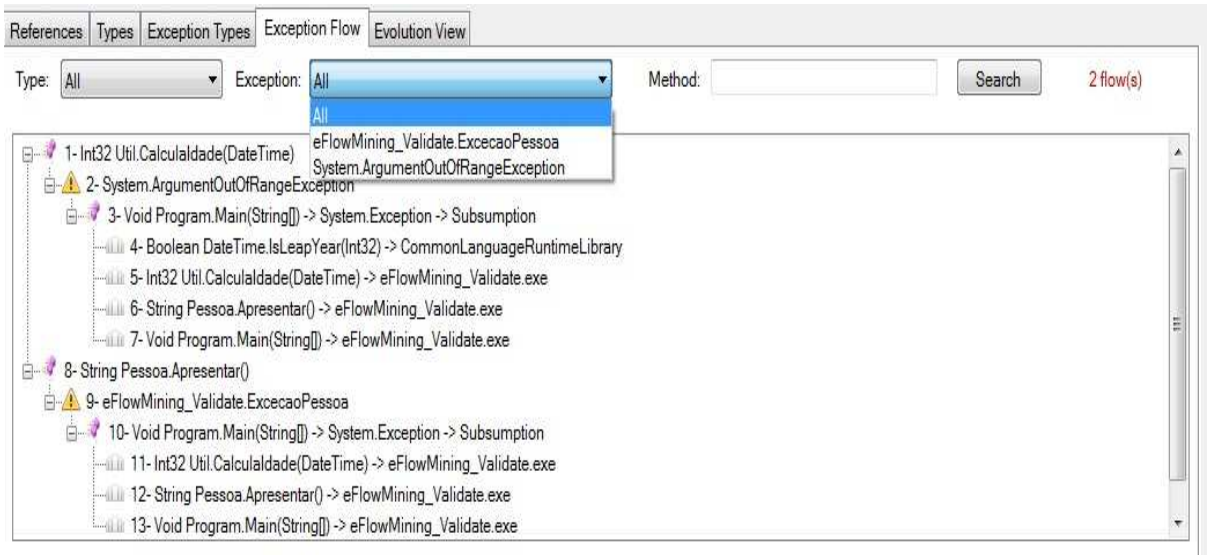
Figura 13 - Visão *Evolution*

Outro cenário pode ser visualizado para o método `Pessoa.Apresentar()`. Observa-se que a versão 1.0.0.0 possuía uma exceção lançada e não tratada, já nas versões 2.0.0.0 e 3.0.0.0 observa-se que a mesma foi tratada, mas não é possível saber como. Para tal, basta um duplo clique no método que será mostrado a visão *Exception Flow*. Quando a visão *Exception Flow* é acessada a partir da visão *Evolution*, a árvore lista para cada versão os fluxos excepcionais provenientes do método e destaca quando houver mudança entre uma versão e outra. Por exemplo, a Figura 14 mostra a forma como o fluxo do método `Pessoa.Apresentar()` foi tratado entre as três versões. O tipo da exceção lançada para as três versões foi a `ExcecaoPessoa` e o método final da pilha de chamadas em que a mesma deveria ter sido tratada foi o `Program.Main`. Para a versão 1.0.0.0 observa-se que não houve tratamento, para a versão 2.0.0.0 foi tratado de forma genérica pelo supertipo `System.Exception`, e para a versão 3.0.0.0 foi tratado de forma especializada pelo mesmo tipo (`ExcecaoPessoa`).

Figura 14 - Fluxos do método `Pessoa.Apresentar()`

Quando a visão *Exception Flow* é acessada diretamente pela aba inferior da tela de resultados, o desenvolvedor tem a possibilidade de filtrar o tipo de tratamento: *Specialized*, *Subsumption* e *Uncaught*; pelos tipos exceções lançadas e pelo nome do método. Conforme pode ser visto na Figura 15, essa visão exibe os fluxos na seguinte ordem em forma de árvore: (i) o método em que a exceção foi lançada, (ii) o tipo da exceção lançada, (iii) o método final da pilha de chamadas e o respectivo tratamento, e (iv) a pilha de chamadas com o respectivo assembly proprietário de cada método. Note que cada item da árvore é numerado, facilitando assim a visualização. Por exemplo, a Figura 14 mostra um fluxo com a exceção `System.ArgumentOutOfRangeException` sendo lançada pelo assembly externo `ComomLanguageRuntimeLibrary` atravessando três métodos: `Util.CalculaIdade`, `Pessoa.Apresentar` e `Program.Main`, e ao final classificada como *Subsumption*. Seguindo a numeração vemos que esse fluxo ocorreu do item um ao sete.

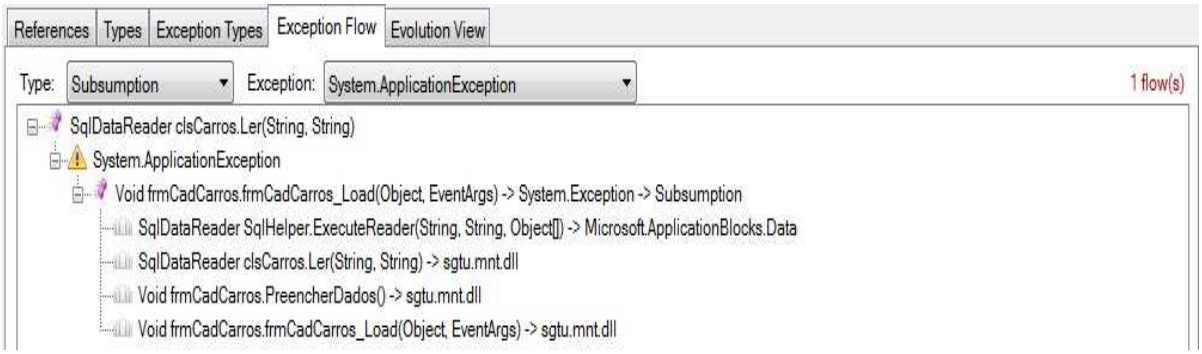
Figura 15 - Visão *Exception Flow*



Além disso, a visão *Exception Flow* suporta a análise só fluxo excepcional entre liguagens de programação diferentes. Para exemplificar, a Figura 16 exibe a *Exception Flow View*, na qual pode-se ver um fluxo excepcional surgindo em um módulo escrito em C# e capturado em um módulo escrito em VB.NET. O método `SqlHelper.ExecuteReader` gera uma exceção do tipo `ApplicationException` no módulo `Microsoft.ApplicationBlocks.Data.dll`, que é capturado pelo método `frmCadCarros.frmCadCarros_Load` do módulo `sgtu.mnt.exe`. Como a exceção

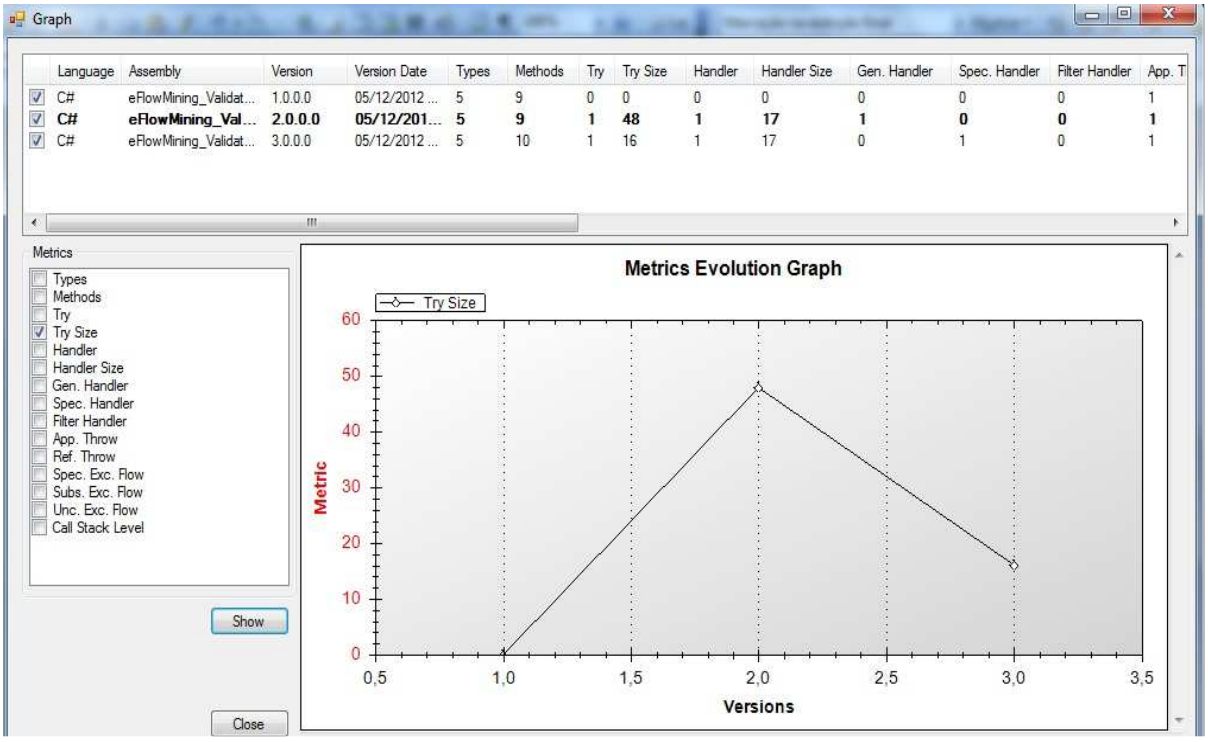
tratada no módulo escrito em VB.NET foi de um super-tipo da exceção gerada no módulo escrito em C#, a ferramenta classificou esse fluxo como *Subsumption*.

Figura 16 - Fluxo excepcional entre linguagens



Para finalizar, a visão *Graph* é exibida na Figura 17. Nela é possível visualizar a evolução dos valores das métricas através de um gráfico de linhas. Por exemplo, no gráfico mostrado na Figura 17 é possível identificar, nas versões 1, 2 e 3, uma variação considerável na métrica *Try Size*. Deste modo, a *Graph View* ajuda os desenvolvedores .NET a terem uma visão geral sobre como a aplicação está implementada no que diz respeito ao tratamento das exceções. O agrupamento de diferentes versões da mesma aplicação também provê a visualização do comportamento das várias métricas durante a evolução da aplicação. Isso permite identificar as mudanças de cenários que tem efeitos positivos ou negativos para o comportamento do tratamento de exceções.

Figura 17 - Visão *Graph*



3.2 Arquitetura

Um dos principais elementos arquiteturais da ferramenta é o framework que permite ler os metadados e o código *IL* de *assemblies* .NET. Três frameworks foram encontrados e analisados: *RAIL - Runtime Assembly Instrumentation Library* (CABRAL; MARQUES; SILVA, 2005), *Mono.Cecil* (EVAIN, 2010) e a *CCI – Microsoft Research Common Compiler Infrastructure Metadata API* (BARNETT; FAHNDRICH; VENTER, 2010).

Desenvolvida pelo departamento de ciência da computação da universidade de Coimbra Portugual, a *RAIL* é uma biblioteca que permite que os *assemblies .NET* sejam lidos e manipulados em tempo de execução. Os elementos internos do *assembly* (tipos, variáveis, métodos, código IL) foram mapeados para uma abstração de objetos de mais fácil entendimento e uso. Seus recursos incluem: (i) alteração de tipos, variáveis, propriedades e chamadas de métodos, (ii) ler e alterar o código IL, (iii) adicionar código antes e após o corpo do método, (iv) cópia de tipos e métodos entre *assemblies*, (v) redirecionar acesso e chamadas a métodos para *proxies*, (vi) criar e salvar *assemblies* em tempo de execução. Ela pode ser utilizada em vários cenários, por exemplo, análise de *assemblies* em tempo de execução, otimização de código *IL*, verificações de segurança, entre outros.

A biblioteca *Mono.Cecil*, que faz parte do projeto *Mono* (ICAZA, 2001), também permite gerar e inspecionar programas no formato *IL*. O projeto Mono é uma iniciativa *open-source* de implementação do .NET Framework para a plataforma Linux. Atualmente ela é utilizada dentro do projeto *Mono* como apoio para algumas ferramentas: *Mono Debugger*, *MoMA* (ferramenta para migração de código .NET Windows para Linux, *DB4O* (banco de dados orientados a objetos), *Gendarme* (inspeção de código fonte .NET com base em boas práticas de programação), entre outras.

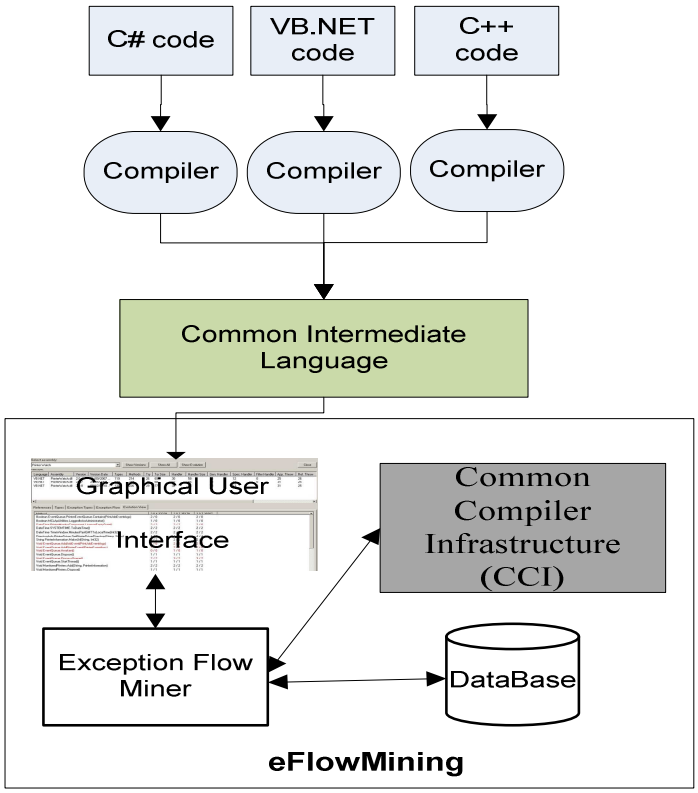
O *Common Compiler Infrastructure (CCI)* é um projeto da *Microsoft Research*. Trata-se de um conjuto de bibliotecas e APIs que suportam algumas funcionalidades que são comuns a compiladores e ferramentas de programação. A API *CCI Metadata* é um subprojeto do *CCI*, que permite analisar ou modificar *.NET assemblies* com promessa de melhor perfomance no uso das classes `System.Reflection` e `System.Reflection.Emit`. Ela pode ser utilizada em vários cenários, por exemplo: compiladores, ferramentas de análise estática e ferramentas de reescrita de código. Como exemplo mais específico de utilização pode-se citar: (i) análise de *.NET assemblies* a procura de práticas de programação ruins e erros em potencial; (ii) leitura dos metadados de um *.NET assembly* e dos comentários de

documentação para criar um arquivo de ajuda de referência; e (iii) inserção de códigos de instrumentação, rastreamento ou segurança dentro do corpo de métodos.

O *framework* escolhido foi o *CCI Metadata* (BARNETT; FAHNDRICH; VENTER, 2010) pelas seguintes razões: (i) a versão 0.5.7 do *framework RAIL* e a 0.9.3 do *Mono.Cecil* foram testadas e apresentaram erros ao tentar carregar um *assembly* .NET; (ii) a ultima versão disponível do *framework RAIL* é de dezembro de 2005; (iii) com a versão 2.0.8 do *framework CCI Metadata*, todos os *assemblies* testados foram carregados com sucesso; (iv) o *framework CCI Metadata* é utilizado pela Microsoft em vários produtos, isto garante certo nível de continuidade na sua atualização.

Após escolha do *framework* que serviria de base para a coleta das informações excepcionais, a arquitetura foi definida. Tal arquitetura é organizada em quatro componentes centrais: Interface Gráfica com o Usuário (GUI), *Exception Flow Miner*, CCI e o Banco de dados. A Figura 18 a exibe.

Figura 18 - Componentes da arquitetura da *eFlowMining*



O módulo *Exception Flow Miner* recebe do componente GUI um *assembly* .NET e usa as interfaces CCI (tal como `IAssembly`, `INamedTypeDefinition`, `IMethodDefinition` e `IOperation`) para realizar o processamento do fluxo

excepcional. Esse processamento primeiramente constrói o gráfico de chamadas do *assembly*, que será utilizado pelo algoritmo *eFlowMining* para a determinação dos fluxos excepcionais e coleta das métricas. A precisão do nosso gráfico de chamadas é baseada nas informações de tipos providas pelo *CCI Metadata*. As métricas e fluxos obtidos são armazenados em um banco de dados relacional para que, a partir de uma combinação de consultas SQL, sejam recuperadas na interface com o usuário.

A escolha por utilizar banco de dados foi motivada principalmente necessidade de avaliar as mudanças sofridas pelo comportamento excepcional durante a evolução da aplicação. Além disso, percebeu-se que as ferramentas existentes não possuíam mecanismos de persistência dos dados, ficando a análise e exibição dos resultados sempre condicionada ao processamento dos aplicativos que estavam sob análise, ou seja, o registro histórico das análises para futuras comparações simplesmente não existia. Sendo assim, o uso de banco de dados foi definido como premissa primordial na arquitetura da *eFlowMining*, permitindo recuperação rápida do histórico das análises realizadas, bem como de visões importantes como a *Evolution View* apresentada na seção 3.1.

Além dessa motivação baseada nos anseios dos desenvolvedores, outro aspecto que foi levado em conta foi a utilização da ferramenta pela indústria (AYEWAH; PUGH, BESSEY, 2010). A arquitetura proposta permite um servidor de banco de dados centralizado e vários clientes processando e visualizado os resultados. Sendo assim, um processo de desenvolvimento exeqüível com a *eFlowMining* poderia ser definido da seguinte forma: o gerente de configuração antes de liberar qualquer versão de um software em produção, efetuaria o processamento do mesmo na *eFlowMining* e avisaria a área de testes para que os resultados fossem analisados. A área de testes por sua vez, daria o *feedback* para o setor de desenvolvimento tratar exceções que vazaram, eliminar fluxos excepcionais que surgiram entre uma versão anterior e a atual, entre outras ações corretivas. O ciclo assim seria reiniciado, só que dessa vez com uma probabilidade maior de lançamento de release com maior robustez.

3.3 Implementação

O primeiro passo do algoritmo de processamento é carregar o *assembly* na memória, percorrer e armazenar no banco de dados seus tipos, métodos e instruções *IL* de cada método. Para tal foi utilizada as seguintes interfaces e métodos do *CCI Metadata*:

- `IAssembly` representando o *assembly*.
- `INamedTypeDefinition` representando um tipo do *assembly*.
- `IMethodDefinition` representando um método de um tipo.
- `IOperation` representando uma instrução *IL*.
- `IAssembly.GetAllTypes` para retornar os tipos de um *assembly*.
- `INamedTypeDefinition.Methods` para retornar os métodos de um tipo
- `IMethodDefinition.Body.Operations` para retornar as instruções *IL* de um método.

O segundo passo é contabilizar as informações do comportamento excepcional de cada método. Tal informação é disponibilizada no código *IL* na forma de registros de uma tabela. Esses registros identificam: a instrução inicial e a final do bloco de código protegido pela clausula `try`; a presença de tratadores para os referidos blocos; o tipo da exceção que está sendo tratada; e a instrução inicial e final de cada tratador. Em *Java* é possível saber que exceções o método pode lançar, porém em *.NET* é necessário fazer uma detalhada análise estática por todos os métodos que são invocados em algum momento, partindo do método que originou a chamada (CABRAL; SACRAMENTO; MARQUES, 2007). A interface utilizada na ferramenta para recuperar as informações de *EH* de cada método foi a `IOperationExceptionInformation` e o método `IMethodDefinition.Body.OperationExceptionInformation`. Nesse momento as métricas *Try*, *Catch*, *Catch Generic e Catch Specialized* são armazenadas. Para definir se um tratador é genérico, a ferramenta compara se o tipo da exceção de cada cláusula `Catch` é igual a qualquer uma das seguintes classes do *.NET*: `System.Exception`, `System.SystemException` ou `System.ApplicationException`.

A métrica *Throw* também é contabilizada nesse passo, porém para tal é necessário analisar cada instrução *IL* através da interface `IOperation`. Essa interface possui o atributo `OperationCode` que retorna todos os tipos de instruções *IL*, por exemplo: `Call`, `Callvirt`, `Newobj`, `Throw`, etc. Quando o `OperationCode` de duas instruções seguidas são respectivamente `Newobj` e `Throw`, a ferramenta contabiliza a métrica.

O terceiro passo é registrar toda a cadeia de chamadas a métodos do *assembly*. Um método é unicamente identificado no banco de dados. Durante a varredura das instruções de um método, a ferramenta detecta uma chamada a outro método através do `OperationCode Call` ou `Callvirt`. Quando isso acontece, o método invocado, suas informações do

comportamento excepcional e a posição do código em que a chamada foi feita são registradas no banco de dados, estando esse método localizado no próprio assembly que está sendo analisado, localizado no *.NET Framework* ou em algum *assembly* referenciado. Dessa forma é possível saber quais exceções estão sendo geradas fora do domínio do *assembly* principal e se as chamadas estão localizadas dentro do *EHC*.

O quarto e último passo é percorrer a cadeia de chamadas a fim de identificar os fluxos excepcionais. Inicialmente é criada uma lista com todos os métodos, do *assembly* principal e suas referências, que lançam exceções. A partir daí, do nó mais inferior até o maior nível da cadeia de chamadas, a ferramenta vai recursivamente percorrendo os métodos e checando se a chamada ao método estava dentro de um bloco `try`. Se estiver, a ferramenta realiza uma comparação da classe da exceção lançada com a classe da exceção de cada tratador. Para tal é utilizado o método `Type.GetType(string TypeName)`. Se as classes forem diferentes, ainda é realizado teste para identificar *subsumption*. Nesse caso é utilizado o método `Type.IsAssignableFrom(Type c)` para determinar se a exceção lançada é sub-classe da exceção do tratador. Se em toda cadeia de chamadas não houver um bloco `try` dentro do contexto da chamada do método, então a métrica *Uncaught* é contabilizada, caso contrário as métricas *Subsumption* e *Specialized* são contabilizadas de acordo com as regras acima.

O diagrama relacional do banco de dados que é populado durante os passos explicados acima é exibido na Figura 19. A Tabela 9 lista as entidades com uma breve descrição sobre as mesmas.

Figura 19 - Diagrama relacional do banco de dados da *eFlowMining*

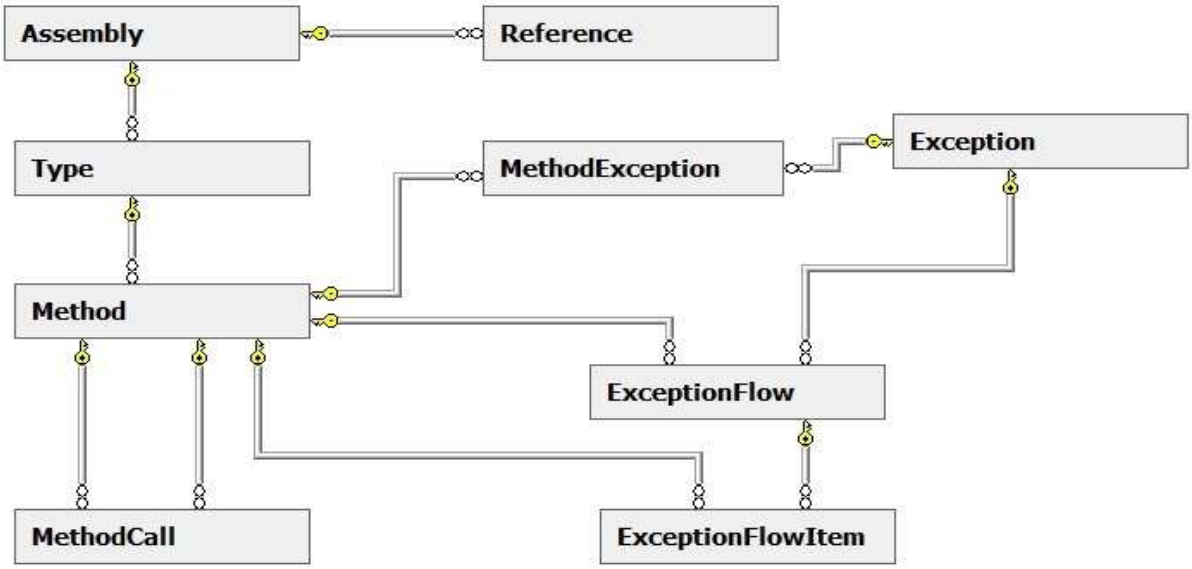


Tabela 9 - Entidades do banco de dados da *eFlowMining*

Entidade	Descrição
`Assembly`	Contêm informações sobre o *assembly* que está sendo analisado, bem como suas referências. Cada assembly é unicamente identificado a partir da versão.
`Reference`	Contêm as referências a *assemblies* externos e aos do *.NET Framework.*
`Type`	Contêm os tipos (classes, interfaces, classes abstratas etc) dos *assemblies* envolvidos no processamento.
`Method`	Contêm os métodos de um tipo.
`MethodCall`	Contêm o fluxo de chamadas dos métodos com a posição do código em que o método de origem chamou o método destino.
`Exception`	Contêm os diversos tipos de exceções lançadas e capturadas.
`MethodException`	Contêm informações sobre o comportamento excepcional de um método. Quantos e em que região do código estão os blocos `try/catch` e a cláusula `throw`.
`ExceptionFlow`	Contêm o resumo dos fluxos excepcionais. Com o método que lançou a exceção e o último método da pilha de execução, tendo ele capturado ou não a exceção.
`ExceptionFlowItem`	Contêm a pilha de execução de um fluxo excepcional.

4 AVALIAÇÃO

Com intuito de avaliar se o uso da ferramenta *eFlowMining* pode ajudar a melhorar a robustez de sistemas de software compilados para a plataforma .NET foram realizadas duas avaliações. A seção 4.1 descreve uma avaliação da compatibilidade e da precisão da ferramenta em relação às diferentes linguagens de programação suportadas pela plataforma .NET. Por sua vez, a seção 4.2 avalia se a ferramenta consegue identificar possíveis defeitos durante a evolução de sistema reais de software.

4.1 Avaliação da Precisão e Compatibilidade

Essa avalição teve o objetivo de verificar a compatibilidade e a precisão da ferramenta em relação às diferentes linguagens de programação suportadas pela plataforma .NET. Foram escolhidas cinco aplicações .NET de diferentes linguagens para execução da ferramenta, validação e interpretação das métricas coletadas. As métricas ajudam a apontar para um bom ou mau uso dos mecanismos de tratamento de exceções, o que resultaria em aplicações mais ou menos robustas. Aspectos como negligenciamento do tratamento de exceções, tratamento por *subsumption*, quantidade de código excepcional do tratador e vazamento de exceções podem ser deduzidas a partir das métricas coletadas.

As aplicações escolhidas das cinco linguagens da plataforma .NET mais utilizadas foram: *PrinterWatch*[1], uma biblioteca de classes VB.NET para monitoramento de uma ou mais impressoras; *DotNetZip*[2], uma biblioteca de classes C# para manipulação de pastas e arquivos ZIP; *CharLS*[3], uma implementação otimizada do padrão JPEG-LS para compressão de imagens sem perdas ou quase sem perdas; *Storm*[4], uma ferramenta F# código aberto para teste de web services; *JUnit*[5], um *framework* em J# para teste unitário. A Tabela 10 exibe as métricas coletadas após a execução da ferramenta.

[1] http://printqueuewatch.codeplex.com
[2] http://dotnetzip.codeplex.com/
[3] http://charls.codeplex.com/
[4] http://storm.codeplex.com/
[5] http://www.junit.org/

Tabela 10 - Métricas coletadas em aplicações multi-linguagem

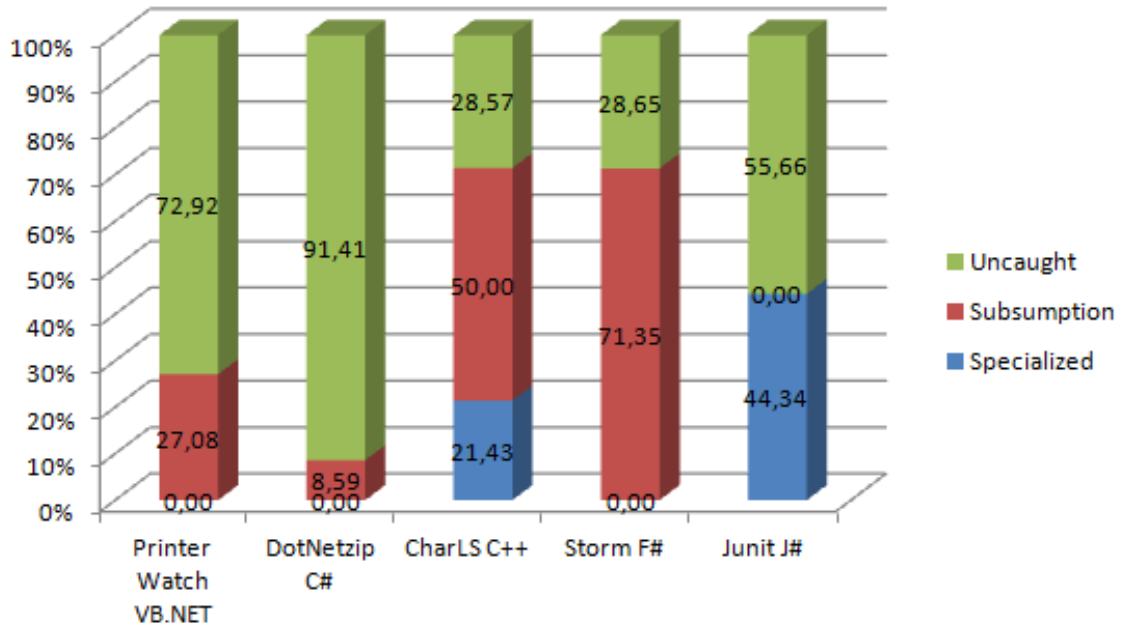
Métrica	Printer Watch VB.NET	DotNet Zip C#	CharLS C++	Storm F#	JUnit J#
Types	119	90	708	1750	104
Methods	214	593	1458	6553	623
Try	26	25	45	18	22
Try Size	69	96	32	43	25
Handler	30	28	37	26	34
Handler Size	58	39	20	28	22
Generic Handler	18	6	31	26	1
Specialized Handler	12	22	6	0	33
Filter Handler	0	0	0	0	0
Application Throw	25	204	8	419	27
Reference Throw	26	44	0	117	59
Specialized Exception Flow	0	0	15	0	98
Subsumption Exception Flow	13	47	35	625	0
Uncaught Exception Flow	35	500	20	251	123
Call Stack Levels	3	5	6	4	3

A análise sobre os resultados da Tabela 10 é listada a seguir:

- As métricas *Generic Handler, Filter Handler* and *Subsumption Exception Flow* podem ser usadas para identificar a ocorrência de exceções capturadas por *subsumption* ou quando filtros forem aplicados. Por exemplo, na aplicação CharLS e Storm, a maiorias das exceções são capturadas por subsunção.
- A métrica *Try Size* pode indicar uso inadequado de blocos try. O resultado para o *assembly DotNetZip* mostra que essa aplicação tem uma média de 96 instruções *IL*. Esse alto espoco de tratamento pode ser a causa do alto número de *Uncaught Exception Flow* para essa aplicação.
- A métrica *Handler Size* permite a identificação de exceções desprezadas. Por exemplo, uma média baixa de instruções *IL* dentro de um tratador pode revelar que alguns tratadores estão vazios ou não estão tratando as exceções.
- A métrica *Reference Throw* expõem a quantidade de exceções lançadas por bibliotecas de terceiros que são geralmente escondidas por interfaces de exceções imprecisas ou documentação falha.
- A métrica *Call Stack Levels* pode revelar alguns desvios na ligação de tratador semi-dinâmica. Uma alta média na quantidade de *Call Stack Levels* pode indicar que o tratamento de exceções está pouco significativo ou com dificuldade de depuração.

- A métrica *Uncaught Exception Flow* pode indicar a presença de possíveis falhas uma vez que exceções criadas pela aplicação ou aplicação de terceiros não estão sendo tratadas, ou seja, estão vazando.
- A quantidade maior de exceções da métrica *Specialized Exception Flow* da aplicação da linguagem J# com relação às outras aplicações denota o suporte a exceções checadas dessa linguagem. Vide seção 2.5.2.
- A quantidade de exceções não tratadas ou tratadas de forma genérica é maior que a quantidade de exceções tratadas de forma especializada em todas as aplicações. Esse comportamento pode ser visto na Figura 20, por exemplo, as aplicações *PrinterWatch*, *DotNetZip* e *Storm* não possuem fluxos excepcionais do tipo *Specialized*. Isso indica pouca robustez dessas aplicações, pois vários problemas ou não são tratados ou são tratados de forma descuidada.

Figura 20 - Percentual dos fluxos excepcionais



Um aspecto que precisa ser levado em consideração com relação à coleta das métricas apresentadas e interpretadas acima é a imprecisão das ferramentas de análise estática, conforme visto na seção 2.2. Para a *eFlowMining* foi feito um trabalho de validação das métricas coletadas e descoberto situações de falso positivo e falso negativo em métodos polimórficos.

Para exemplificar a situação acima, incluímos a subclasse `Aluno` no programa exemplo utilizado na seção 3.1 e sobrescrevemos o método `Apresentar`. O novo diagrama

de classe pode ser visto na Figura 21 e o código da classe que ilustra essa situação é exibido na Figura 22.

Figura 21 - Diagrama de classes do programa exemplo com classe `Aluno`

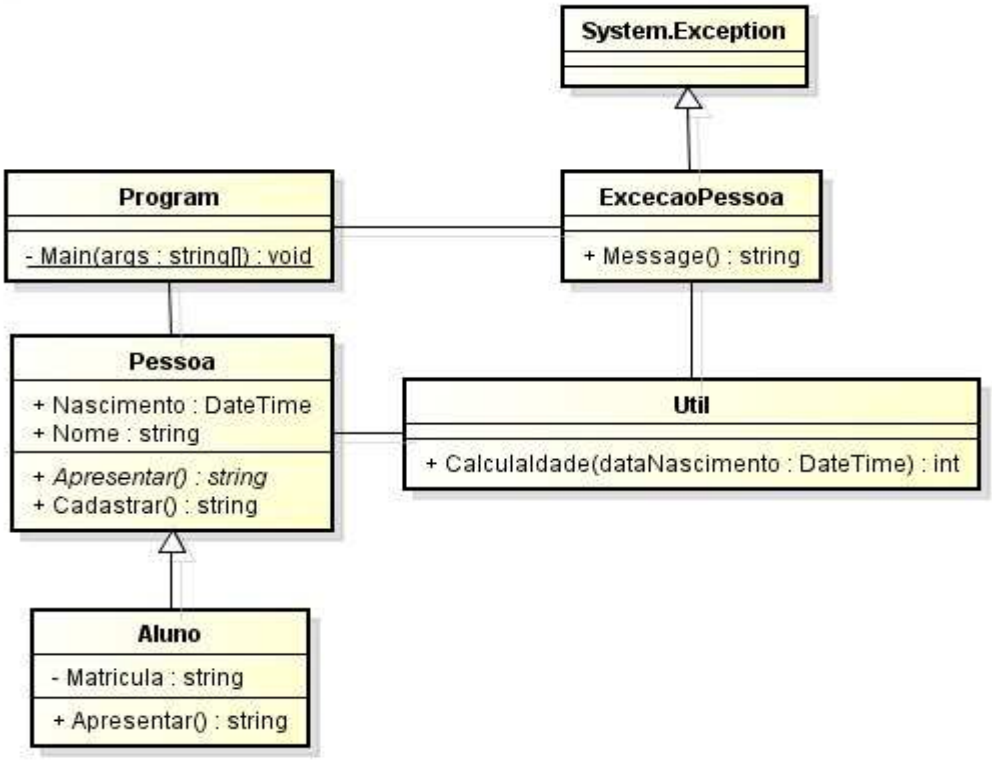
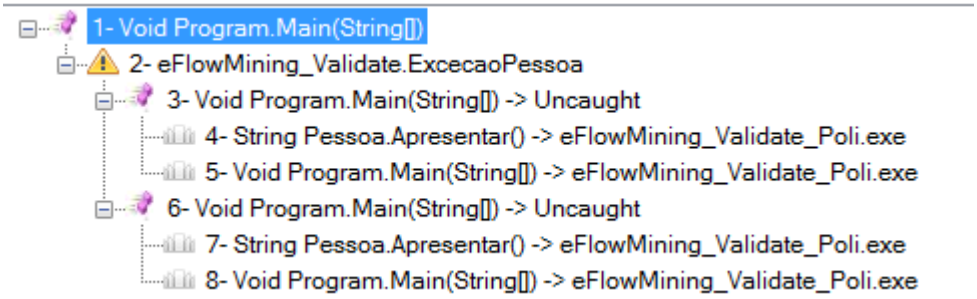


Figura 22 - Código com a limitação do polimorfismo

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace eFlowMining_Validate
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             Pessoa pessoa = new Pessoa() { Nome = "Pessoa 1" };
13             Aluno aluno = new Aluno() { Nome = "Aluno 1", Matricula = "123" };
14
15             Console.WriteLine(pessoa.Apresentar());
16             Console.WriteLine(aluno.Apresentar());
17
18             Console.ReadLine();
19         }
20     }
21 }
22
```

A classe `Aluno` é instanciada na linha de código 13 e o método `Apresentar` é invocado na linha 16, a *eFlowMining* porém, não irá verificar o código do método `Apresentar` da classe `Aluno`, ela irá verificar o código da superclasse `Pessoa`, pois a instrução *IL* do código objeto para a linha 16 faz referência a ela e não a subclasse `Aluno`. Desta forma, os possíveis fluxos oriundos do método `Apresentar` da subclasse `Aluno` serão ignorados, sendo um exemplo de falso negativo, além de também representar uma situação de falso positivo, pois irá reportar dois fluxos para o método `Apresentar` da superclasse `Pessoa`, quando na verdade somente um será executado. A Figura 23 ilustra esses dois fluxos excepcionais. Os itens 4 e 7 mostram que o método verificado foi o `Apresentar` da classe `Pessoa` e não da classe `Aluno`.

Figura 23 - Fluxo excepcional com a limitação do polimorfismo



Apesar dessa limitação e da imperfeição intriseca das ferramentas de analise estática (vide item 2.2), consideramos que as métricas coletadas foram úteis para a interpretação do comportamento excepcional das aplicações avaliadas.

4.2 Identificação de defeitos durante a evolução de sistemas reais software

Após verificar a compatibilidade e a precisão da ferramenta na avaliação inicial, iremos avaliar se a *eFlowMining* ajuda a identificar defeitos durante a evolução de sistemas de software. Assim, o objetivo é avaliar se a visão histórica dos dados permite identificar com maior facilidade problemas existentes em várias versões de aplicações reais. Usaremos como base as aplicações e a metodologia do estudo realizado por Cabral e Marques (2007), que as dividiu de acordo com sua natureza em quatro categorias: bibliotecas de classes, aplicações ASP.NET, softwares que rodam como serviço em servidores e aplicações *desktop*. Para cada categoria, ele escolheu quatro aplicações. Como algumas das aplicações desse estudo não possuíam mais código fonte e binário disponíveis e outras não possuíam registro

histórico de erros entre versões, decidimos escolher pelo menos uma aplicação de cada categoria, exceto pela categoria software que roda em servidor, que não foram encontradas aplicações, sendo assim foi escolhida uma aplicação dessa categoria no *website* de compartilhamento de código *codeplex.com*. Além disso, essas aplicações foram escolhidas pelo fato do estudo de Cabral e Marques (2007) ter identificado problemas no modo os desenvolvedores usaram os mecanismos de tratamento de exceções, tais como: (i) lançamento de exceções genéricas que tornam praticamente impossível que se trate e se recupere o erro sem o encerramento do sistema, (ii) captura de exceções genéricas sem o devido tratamento, fazendo com que o software continue a ser executado em estado corrompido, e (iii) ausência de código relativo a tratamento de exceções (lançamento e/ou captura). No entanto, esses problemas foram identificados para a versão analisada na época do estudo. Como na nossa avaliação iremos analisar a evolução dessas aplicações, poderemos verificar se os problemas persistiram até a ultima versão disponível e se os dados apresentados pela *eFlowMining* ajudaram a identificar defeitos reais propagados entre as diversas versões das aplicações selecionadas.

A metodologia desta avaliação seguirá os seguintes passos:

1. Escolha de uma aplicação de cada categoria do estudo do Cabral e Marques (2007) que possua versões disponíveis para download e registro histórico detalhado de erros.
2. Processamento de todas as versões disponíveis das aplicações selecionadas na *eFlowMining*.
3. A partir da visão *Evolution* da *eFlowMining*, pré-selecionar métodos que tiveram variação na quantidade de fluxos excepcionais que os mesmos originaram durante a evolução da aplicação. Como variação entende-se por:
    - Fluxo excepcional novo e não tratado;
    - Fluxo excepcional novo e tratado de forma genérica;
    - Fluxo excepcional novo e tratado de forma especializada;
    - Desaparecimento de fluxo excepcional.
4. A partir da lista de métodos do passo anterior, selecionar os métodos em que os fluxos excepcionais foram gerados dentro da própria aplicação (vide Tabela 7), ou caso não existam, só fluxos excepcionais quem sejam indicativos de um problema real de acordo com a natureza de cada aplicação.

5. Analisar o código fonte dos métodos e fluxos excepcionais selecionados de todas as versões a fim de identificar se houveram ou não mudanças no código fonte que resolviam os defeitos encontrados.
6. Comparar a análise do código fonte com o registro de erros de cada aplicação para identificar se a ferramenta apontou cenários de falhas que não foram identificados pelos desenvolvedores.

As aplicações escolhidas foram: *AscGen*[6], uma aplicação desktop que converte imagens em texto ASCII; *PhotoRoom*[7], um *website* ASP.NET para gerenciamento de álbuns de foto e galeria de imagens; *Report.NET*[8], uma biblioteca de classes para geração de arquivos PDF; e *SuperWebSocket*[9], uma implementação em .NET de um servidor *WebSocket*. A Tabela 11 exibe as aplicações e as versões disponíveis que foram processadas e os tipos de exceções descartadas das mesmas.

Tabela 11 - Aplicações/versões selecionadas

Aplicação	Número das versões	Qtd.	Exceções descartadas
AscGen	0.7.1, 0.7.2, 0.7.3, 0.8.0, 0.8.1, 0.8.2, 0.9.0, 0.9.1, 0.9.5, e 0.9.6	10	`System.Exception`, `FormatExcepcion`, `ArgumentException`, `ArgumentNullException`, `ArgumentOutOfRangeException`
PhotoRoom	1.0, 1.4, 1.5, 1.6 e 1.7	5	Nenhuma
Report.NET	0.7.13, 0.7.14, 0.8.0, 0.8.1, 0.9.0, 0.9.1, 0.9.2, e 0.9.5	8	`FormatExcepcion`, `ArgumentException`, `ArgumentNullException`, `ArgumentOutOfRangeException`
SuperWebSocket	1, 2, 3, 4, 5 e 6	6	Nenhuma

Os métodos e fluxos excepcionais selecionados no passo quatro da metodologia são apresentados em tabelas individuais para cada aplicação. Para melhor visualização, os números das versões serão substituidos pelo número de ordem durante a evolução da aplicação. As cores contidas nas colunas de cada versão estão definidas da seguinte forma: vermelho, fluxo não tratado; verde, fluxo tratado de forma genérica; azul, fluxo tratado de forma especializada.

---

[6] http://sourceforge.net/projects/ascgen2

[7] http://sourceforge.net/projects/photoroom

[8] http://sourceforge.net/projects/report

[9] http://superwebsocket.codeplex.com

Tabela 12 - Fluxos excepcionais escolhidos para a aplicação AscGen

Aplicação: AscGen		Versões									
Método	Exceção	1	2	3	4	5	6	7	8	9	10
ImageToColors.Convert	UnathorizedAccessException										
Version.GetVersion	OutOfMemoryException										
FormBatchConvert.ConversionThread	OutOfMemoryException										
TextToImage.Save	ComponentModel.InvalidEnumArgumentException										
ImageToValues.Convert	ComponentModel.InvalidEnumArgumentException										

Tabela 13 - Fluxos excepcionais escolhidos para a aplicação PhotoRoom

Aplicação: PhotoRoom		Versões				
Método	Exceção	1	2	3	4	5
DirectoryLister.Compare	ArgumentException					
Config.SetConfigSettings	ArgumentNullException					
Config.SetConfigSettings	FormatException					
Slideshow.RefreshRate_SelectedIndexChanged	ArgumentException					

Tabela 14 - Fluxos excepcionais escolhidos para a aplicação Report.NET

Aplicação: Report.NET		Versões							
Método	Exceção	1	2	3	4	5	6	7	8
RT.PrintPDF	Reports.ReportException								
RT.ViewPDF	Reports.ReportException								
RT.ResponsePDF	Web.HttpException								
RT.ResponsePDF	Reports.ReportException								
RT.ResponsePDF	UnathorizedAccessException								
RT.StartAcro	InvalidOperationException								

Tabela 15 - Fluxos excepcionais escolhidos para a aplicação SuperWebSocket

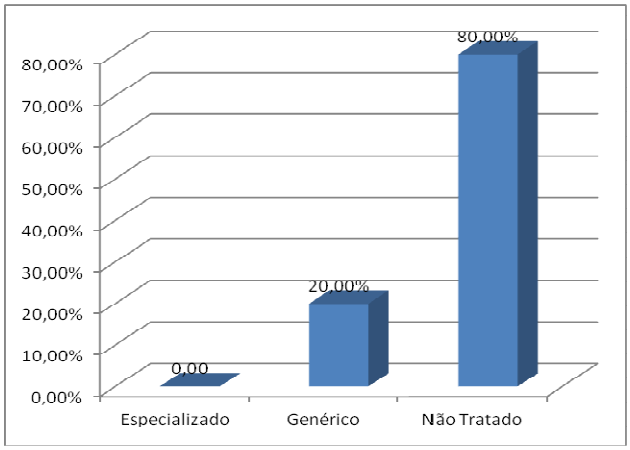
Aplicação: SuperWebSocket		Versões					
Método	Exceção	1	2	3	4	5	6
WebSocketDataReader.FindCommand	ArgumentNullException						
WebSocketDataReader.FindCommandInfo	ArgumentNullException						
WebSocketHeaderReader.FindCommand	ArgumentNullException						
WebSocketHeaderReader.FindCommandInfo	ArgumentNullException						
DraftHybi00Processor.GetResponseSecurityKey	ObjectDisposedException						

Alguns tipos de exceções descartados em uma aplicação foram selecionados em outra. Por exemplo, as exceções `ArgumentException`, `ArgumentNullException` e a `FormatException` foram descartados para as aplicações *AscGen* e *Report.Net*, pelo fato

dessas aplicações possuírem outros tipos de exceções mais relacionados com sua natureza. O software *AscGen* trabalha com manipulação de imagens e com arquivos, por isso as exceções `UnathorizedAccessException`[10], `OutOfMemoryException`[11] e `ComponentModel.InvalidEnumArgumentException`[12] foram escolhidas, conforme Tabela 12. Já a biblioteca de geração de arquivos PDF *Report.NET*, além de ser a única a possuir tipo de exceção específica (`Reports.ReportException`) e lançada dentro do próprio *assembly*, possui também exceções relacionadas com suas funcionalidades, tais como: `Web.HttpException`[13] e `InvalidOperationException`[14]. Para os softwares *PhotoRoom* e *SuperWebSocket* foram selecionados todos os fluxos excepcionais do passo três da metodologia, nenhum tipo de exceção foi descartado.

Em resumo, nenhum fluxo excepcional foi tratado de forma especializada, apenas quatro foram tratados de forma genérica na última versão, e a maioria não foram tratados. Esse comportamento pode ser visualizado no gráfico da Figura 24.

Figura 24 - Distribuição dos tipos de fluxos excepcionais selecionados



[10] UnathorizedAccessException: é lançada quando o sistema operacional nega acesso a algum recurso de I/O.

[11] OutOfMemoryException: é lançada quando não há mais memória disponível para continuar com a execução do programa.

[12] ComponentModel.InvalidEnumArgumentException: é lançada quando um componente recebe um valor inválido.

[13] Web.HttpException: é lançada quando ocorre erro no processamento de requisições HTTP.

[14] InvalidOperationException: é lançada quando uma chamada a método é inválida para o estado atual do objeto.

Com exceção de dois fluxos excepcionais do software *SuperWebSocket*, todos os outros mantiveram o comportamento desde a versão em que eles se originaram até a última. Após análise no código fonte desses dois fluxos que desapareceram entre a versão quatro e cinco, foi descoberto que na realidade as assinaturas dos métodos mudaram, mas a exceção permaneceu lá e sem tratamento. O método `WebSocketDataReader.FindCommand` mudou para `WebSocketDataReader.FindCommandInfo` e o método `WebSocketHeaderReader.FindCommand` mudou para `WebSocketHeaderReader.FindCommandInfo`.

Os problemas encontrados até esta fase da avaliação confirmam que os problemas relatados por Cabral e Marques (2007) foram propagados durante a evolução das aplicações selecionadas.

O passo seguinte foi a análise do código fonte dos fluxos selecionados e comparação com o registro de erros de cada aplicação. As tabelas abaixo exibem os resultados encontrados para cada aplicação. A coluna *BugReport* indica se o defeito foi registrado e o *link* desse registro, a coluna *Análise do Código* descreve um cenário de problema que o tratamento realizado ou a falta dele pode acarretar, e a última indica a gravidade do problema para o usuário e será exibida em cores de acordo com a seguinte categorização: vermelha, alta; amarelo, moderada; e verde, baixa. Essa categorização foi definida de acordo com as seguintes regras: alta, indica um problema que o usuário não consegue identificar a origem com os dados apresentados pelo tratamento excepcional, que inviabilize a abertura ou ocasione fechamento abrupto da aplicação; moderada, indica problema que só ocorre quando o usuário usa o software incorretamente; e baixa, indica problema difícil de ocorrer ou que o tratamento exibe informações de indiquem a origem do problema.

Tabela 16 - Análise dos fluxos excepcionais escolhidos para a aplicação AscGen

Aplicação: AscGen				
Método	Exceção	BugReport	Análise do código	
ImageToColors. Convert	Unathorized AccessException	Problema relacionado em sourceforge.net/p/ ascgen2/bugs/8	Esta exceção pode ser lançada na tentativa de exclusão de um arquivo no disco. Pode estar relacionado ao controle de permissões de acesso ao disco da máquina do usuário. O tratamento genérico não indica nada a respeito disso, deixando o usuário confuso.	
Version. GetVersion	OutOfMemory Exception	Sem relato	Esta exceção pode ser lançada na tentativa de montar o texto da versão atual do programa, que é uma das primeiras ações executadas na inicialização. Pode inviabilizar a abertura do software.	
FormBatchConver. ConversionThread	OutOfMemory Exception	Problema relacionado em sourceforge.net/p/ ascgen2/bugs/3	Esta exceção pode ser lançada na montagem do texto do log de conversão da imagem em arquivo ASCII. Pode impedir o funcionamento do principal processo de software.	
TextToImage. Save	ComponentModel .InvalidEnum Argument Exception	Sem relato	Esta exceção pode ser lançada durante o processo de conversão da imagem em texto, ao acessar as bibliotecas gráficas do .NET Framework. Pode impedir o funcionamento do principal processo de software.	
ImageToValues. Convert	ComponentModel .InvalidEnum Argument Exception	Sem relato	Similar ao fluxo excepcional anterior.	

Tabela 17 - Análise dos fluxos excepcionais escolhidos para a aplicação PhotoRoom

Aplicação: PhotoRoom				
Método	Exceção	BugReport	Análise do código	
DirectoryLister. Compare	Argument Exception	Sem relato.	Esta exceção pode ser lançada durante o processo de exibição das pastas do disco para o usuário escolher onde estão suas fotos. Pelo trecho de código analisado, dificilmente irá ocorrer.	
Config. SetConfigSettings	Argument NullException	Sem relato	Esta exceção pode ser lançada durante a escrita da configuração de quantas imagens serão exibidas por álbum. O tratamento genérico empregado define uma quantidade default em caso de erro.	
Config. SetConfigSettings	Format Exception	Sem relato	Similar ao fluxo excepcional anterior.	
Slideshow.Refresh Rate_ SelectedIndex Changed	Argument Exception	Problema relacionado em sourceforge.net/tracker/?func=detail&aid=877249&group_id=54893&atid=475203	Esta exceção pode ser lançada na funcionalidade que modifica o tempo de exibição das fotos no slideshow. Depende de uma entrada errônea do usuário para ocorrer.	

Tabela 18 - Análise dos fluxos excepcionais escolhidos para a aplicação Report.NET

Aplicação: Report.NET				
Método	Exceção	BugReport	Análise do código	
RT.PrintPDF	Reports. ReportException	Problema relacionado em sourceforge.net/tracker/?func=detail&aid=1903766&group_id=58374&atid=487496	Esta exceção pode ser lançada durante o processo de impressão do arquivo PDF, mais especificamente no momento em que é checado se o AdobeAcrobat[15] está instalado. Por estar dentro do próprio assembly, o desenvolvedor controlou as situações em que a mesma é lançada e define mensagens de alerta específicas para o usuário.	
RT.ViewPDF	Reports. ReportException	Problema relacionado em sourceforge.net/tracker/?func=detail&aid=1903766&group_id=58374&atid=487496	Esta exceção pode ser lançada na criação do documento PDF em pasta temporária da maquina do usuário, mais especificamente na checagem se existe outro documento pendente de criação. Por estar dentro do próprio assembly, o desenvolvedor exibe uma mensagem específica para o usuário.	
RT.ResponsePDF	Web. HttpException	Sem relato	Esta exceção pode ser lançada durante o processo de envio do documento PDF para ser visualizado em browser internet, caso o documento esteja corrompido. Para ocorrer depende de problema na criação do documento PDF e como não foi tratada poderá deixar o usuário confuso.	
RT.ResponsePDF	Reports. ReportException	Sem relato	Similar ao método anterior, porém exibe uma mensagem específica para o usuário.	
RT.ResponsePDF	Unathorized AccessException	Sem relato	Similar ao método anterior, porém ocorre em região do código que faz acesso direto a memória RAM. Como não foi tratada poderá deixar o usuário confuso.	
RT.StartAcro	InvalidOperation Exception	Problema relacionado em sourceforge.net/tracker/?func=detail&aid=785521&group_id=58374&atid=487496	Esta exceção pode ser lançada no exato momento em que é enviado um comando para o Windows abrir o AdobeAcrobat. De ocorrência imprevisível e sem tratamento, pode deixar usuário confuso.	

[15] http://get.adobe.com/br/reader

Tabela 19 - Análise dos fluxos excepcionais escolhidos para a aplicação SuperWebSocket

Aplicação: SuperWebSocket				
Método	Exceção	BugReports	Análise do código	
WebSocket DataReader. FindCommand	ArgumentNull Exception	Sem relato	Esta exceção pode ser lançada durante o processo de localização de um comando na stream de bytes recebidos pelo servidor socket, mais especificamente na conversão de byte para string e na classe que trata dos dados principais do protocolo de comunicação. Como o software é um serviço que roda em servidor, ou seja, não tem iteração com usuário final, e a exceção não foi tratada, pode ocasionar em fechamento abrupto do software sem que o usuário perceba.	
WebSocket DataReader. FindCommandInfo	ArgumentNull Exception	Sem relato	Similar ao fluxo excepcional anterior, porém só ocorre a partir da versão cinco.	
WebSocket HeaderReader. FindCommand	ArgumentNull Exception	Sem relato	Similar ao primeiro fluxo excepcional, porém ocorre na classe que trata dos dados de cabeçalho do protocolo de comunicação.	
WebSocket HeaderReader. FindCommandInfo	ArgumentNull Exception	Problema relacionado em superwebsocket.codeplex.com/workitem/11443	Similar ao fluxo excepcional anterior, porém só ocorre a partir da versão cinco.	
DraftHybi00 Processor. GetResponse SecurityKey	ObjectDisposed Exception	Problema relacionado em superwebsocket.codeplex.com/workitem/11443	Esta exceção pode ocorrer durante o processo de *HandShake*[16], mais especificamente durante a checagem da chave de segurança da máquina transmissora. Como a exceção não foi tratada, pode ocasionar em fechamento abrupto do software sem que o usuário perceba.	

Esta análise detalhada do código dos fluxos excepcionais e comparação com registro de erros reais dos usuários dessas aplicações confirmam os problemas relatados pelo estudo de Cabral e Marques (2007), ou seja, os mecanismos de tratamento de exceções foram mal utilizados. O alto número de fluxos excepcionais classificados como de gravidade alta, somado a fato de que mesmo com erros relatados, os fluxos excepcionais continuaram sem

[16] HandShake: é o processo pelo qual duas máquinas afirmar uma a outra que a reconheceu e estão prontas para iniciar a comunicação.

tratamento ou com tratamento genérico até a última versão da aplicação, justificam essa afirmação. A Figura 25 exibe a distribuição percentual da gravidade dos fluxos excepcionais, já a Figura 26 exibe os percentuais de erros relatados.

Figura 25 - Distribuição da gravidade dos fluxos excepcionais

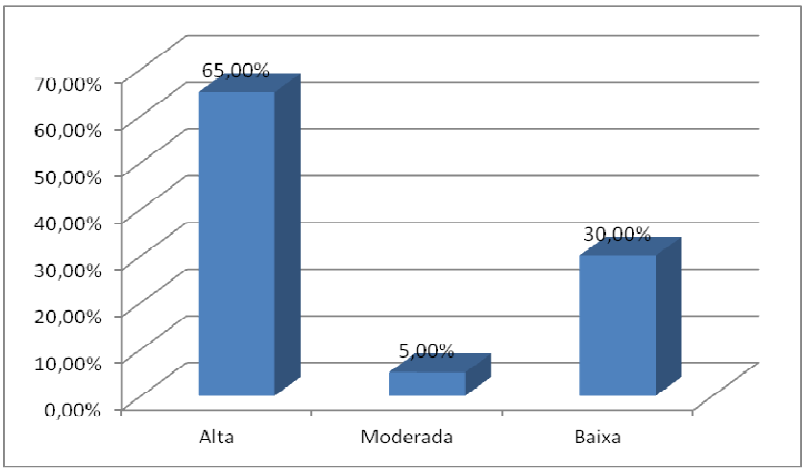
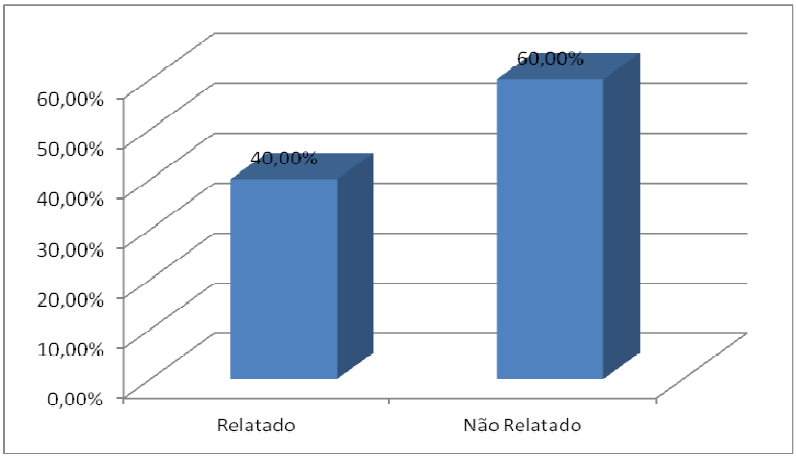


Figura 26 - Distribuição do tipo de relato dos fluxos excepcionais



Essa avaliação mostrou que o uso da *eFlowMining* ajudou a identificar vários defeitos nas aplicações selecionadas e foi mostrado que tais defeitos se transformaram em falhas perceptíveis pelos usuários, tornando as aplicações menos robustas. Por exemplo, dos vinte fluxos excepcionais selecionados, oito foram relatados e doze não foram, sendo que desses oito fluxos somente dois foram tratados e de forma genérica.

5 CONCLUSÃO

Este trabalho apresentou a ferramenta de análise estática do fluxo excepcional *eFlowMining*, cujo objetivo é fornecer aos desenvolvedores .NET uma forma de analisar os fluxos excepcionais de aplicações compiladas para plataforma .NET. Suas funcionalidades foram definidas a partir das limitações das ferramentas de análise estática do fluxo excepcional pesquisadas no estado da arte. Além disso, também foi levado em conta aspectos práticos vivenciados pelos desenvolvedores .NET, tais como: necessidade de armazenamento das informações coletadas em banco de dados e interface gráfica amigável para visualização. Dessa forma os seguintes requisitos foram atendidos pela ferramenta: (i) visualizar as métricas coletadas sobre o tratamento de exceções através de tabelas e gráficos; (ii) visualizar uma representação gráfica do fluxo excepcional em forma de árvore; (iii) acompanhar o comportamento do tratamento de exceções através de várias versões da mesma aplicação; (iv) localizar de forma rápida os tipos de exceções lançadas e seus respectivos tratadores; e (v) consultar o histórico das análises realizadas.

A avaliação teve o intuito de verificar se o uso da ferramenta *eFlowMining* pode ajudar a melhorar a robustez de sistemas de software compilados para a plataforma .NET. Para tal duas avaliações foram realizadas.

A primeira descreveu uma avaliação da compatibilidade e da precisão da ferramenta em relação às diferentes linguagens de programação suportadas pela plataforma .NET. Foram escolhidas cinco aplicações .NET de diferentes linguagens para execução da ferramenta, validação e interpretação das métricas coletadas. A coleta das métricas foi realizada com sucesso para todas as aplicações, e os resultados confirmaram algumas evidencias existentes sobre o tratamento de exceções na plataforma .NET, por exemplo: a quantidade de exceções não tratadas ou tratadas de forma genérica é maior que a quantidade de exceções tratadas de forma especializada em todas as aplicações. Um aspecto de imprecisão da ferramenta relacionado a métodos polimórficos foi encontrado e apresentado nessa avaliação. No entanto, apesar dessa limitação e da imperfeição intrínseca das ferramentas de analise estática, consideramos que as métricas coletadas foram úteis para a interpretação do comportamento excepcional das aplicações avaliadas.

A segunda avaliação verificou se a ferramenta conseguia identificar possíveis defeitos durante a evolução de sistema reais de software. Para tal, sistemas de software reais do estudo realizado por Cabral e Marques (2007) foram selecionados e analisados em todas as versões

disponíveis, segundo metodologia descrita na seção 4.2. Os resultados indicaram que o uso da *eFlowMining* ajudou a identificar vários defeitos nas aplicações selecionadas e foi mostrado que tais defeitos se transformaram em falhas perceptíveis pelos usuários, tornando as aplicações menos robustas.

Como continuidade a pesquisa desenvolvida nessa dissertação pretendemos integrar a ferramenta *eFlowMining* ao ambiente de desenvolvimento oficial da Microsoft para a plataforma .NET, o *Microsoft Visual Studio* ©, na forma de um plugin para uso da indústria, permitindo que o desenvolvedor possa navegar no código fonte dos métodos envolvidos nos fluxos excepcionais apresentados. Além disso, iremos utilizar estudo de clones para identificar mudanças nas assinaturas de métodos a fim de melhorar a precisão da ferramenta.

REFERÊNCIAS

AL-QUTAISH, R. E. **Quality Models in Software Engineering Literature: An Analytical and Comparative Study**. Journal of American Science. 6:(3), 2010, pp. 166-175

AYEWAH, N., PUGH, W. **A report on a survey and study of static analysis users.** In Proceedings of DEFECTS, pages 1-5. 2008.

AYEWAH, N., PUGH, W. **The Google FindBugs Fixit**. In Proceeding of ISSTA, pages 241-252, New York, NY, USA, 2010.

BARNETT, M., FAHNDRICH, M., VENTER, H. **Common Compiler Infrastructure**. 2010. Disponível em: <http://research.microsoft.com/en-us/projects/cci/>.

BEN-ARI, M. **The bug that destroyed a rocket.** In Proceedings of SIGCSE Bulletin. 2001, 58-59.

BESSEY, A. *et al*. **A Few Billion Line of Code Later: Using Static Analysis to Find Bugs in the Real World**. Communications of the ACM, pages 66-75. 2010.

BOEHM, B. W., BROWN, J. R., LIPOW, M. **Quantitative evaluation of software quality.** In Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society, Los Alamitos (CA), USA, 1976; 592-605.

BOEHM, B. W. **Guidelines for Verifying and Validating Software Requirements and Design Specifications**. Samet, P. A., eds.; *Euro IFIP 79*, North Holland (1979), 711-719.

BOOCH, G. **Object-oriented Analysis and Design with Applications, 2nd ed.**, Benjamin/Cummings, 1994.

CABRAL, B., MARQUES, P., SILVA, L. **RAIL: Code Instrumentation for .NET**. In Proc. of the 2005 ACM Symposium On Applied Computing (SAC'05), ACM Press, Santa Fé, New Mexico, USA. 2005.

CABRAL, B., MARQUES, P. **Exception Handling: A Field Study in Java and .NET**. In ECOOP 2007 - 21st European Conference Object-Oriented Programming, vol. 4609 of Lecture Notes in Computer Science, pp. 151–175.

CABRAL, B., SACRAMENTO, P., MARQUES, P. **Hidden truth behind .NETs exception handling today**. IET Software 1(6): 233-250 (2007)

CACHO, N. *et al*. **EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming**. In Proceedings of the 7th international conference on Aspect-Oriented Software Development - AOSD., Pages 72-83. 2008.

CACHO, N. *et al*. **Exception Flows made Explicit: An Exploratory Study**. XXIII Simpósio Brasileiro de Engenharia de Software - SBES'09.

CHANG, B.-M. *et al*. **Interprocedural exception analysis for Java**. In SAC '01: Proceedings of the 2001 ACM symposium on Applied computing, pp. 620–625. ACM, New York, NY, USA.

CHESS, B., WEST J. **Secure programming with static analysis**. Addison-Wesley Professional. 2007.

COELHO, R. *et al*. **Assessing the Impact of Aspects on Exception Flows: An Exploratory Study**. European Conference on Object Oriented Programming (ECOOP 2008). 2008.

EVAIN, J. B. **Mono Cecil**. 2010. Disponível em: <http://www.mono-project.com/Cecil>.

FAHNDRICH, M. *et al*. **Tracking down exceptions in standard ml programs**. Tech. rep., EECS Department, UC Berkeley. 1998.

FU, C., RYDER, B. G. **Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications**. In Proceedings of ICSE'07, pages 230-239, Minneapolis, USA, 2007.

GARCIA, A. *et al*. **A comparative study of exception handling mechanisms for building dependable object-oriented software.** The Journal of Systems and Software, 59(2):pp. 197–222. 2001.

GARVIN, D. **What Does "Product Quality" Really Mean?**. Sloan Management Review. Fall 1984, pp. 25-45.

GOODENOUGH, J. B. **Exception handling: Issues and a proposed notation**. Comm. of the ACM, 18(12):683–696. (1975).

ICAZA, M. **Mono Project**. 2001. Disponível em: < http:// www.mono-project.com/Start>.

ISO. **ISO/IEC 9126-1: Software Engineering - Product Quality - Part 1: Quality Model.** International Organization for Standardization, Geneva, Switzerland, 2001.

JONES, C**. Estimating Software Costs**. McGraw-Hill, New York, 1998.

LANDI, W. **Undecidability of static analysis**. ACM Letters on Programming Languages and Systems (LO-PLAS), 1(4):323 – 337, December 1992.

LANG, J., STEWART, D. B. **A Study of the Applicability of Existing Exception-Handling Techniques to Component-Base Real-Time Software Technology**. ACM Transactions on Programming Languages and Systems, 20(2):pp. 274–301. 1998.

LAPRIE, J.-C., RANDELL, B. **Basic Concepts and Taxonomy of Dependable and Secure Computing.** IEEE Trans. Dependable Secur. Comput., 1(1):pp. 11–33. 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

LEE, P. A., ANDERSON, T. **Fault Tolerance: Principles and Practice. Dependable computing and fault-tolerant systems**. Berlin; New York, 2nd edn.1990

MALAYERI, D., ALDRICH, J. **Practical Exception Specifications**. In Dony, C., Knudsen, J. L., Romanovsky, A. B. and Tripathi, A. (eds.), Advanced Topics in Exception Handling Techniques, vol. 4119 of Lecture Notes in Computer Science, pp. 200–220. Springer. 2006.

MILLER, R., TRIPATHI, A. **Issues with Exception Handling in Object-Oriented Systems.** Lecture Notes in Computer Science, 1241:pp. 85–?? 1997.

PARNAS, D. L., WURGES, H. **Response to undesired events in software systems**. In ICSE '76: Proceedings of the 2nd international conference on Software engineering, pp. 437–446. IEEE Computer Society Press, Los Alamitos, CA, USA. 1976.

PFLEEGER, S.L., ATLEE, J.M. **Software Engineering: Theory and Practice (4th Edition)**. Prentice Hall. 2009.

REIMER, D., SRINIVASAN, H. **Analyzing exception usage in large Java applications.** In Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems, pp. 10–18. 2003.

ROBILLARD, M. P., MURPHY, G. C. **Designing robust Java programs with exceptions**. In Proceedings of the 8th ACM SIGSOFT international Symposium on Foundations of Software Engineering: Twenty-First Century Applications. ACM Press, New York, 2000, 2-10.

ROBILLARD, M. P., MURPHY, G. C. **Static analysis to support the evolution of exception structure in object-oriented systems**. ACM Trans. Softw. Eng. Methodol., 12(2):pp. 191–221. 2003.

ROBINSON, S. *et al*. **Professional C# Third Edition.** Wiley Publishing, Inc., Indianalopis, Indiana, USA. 2004.

RUSTAN, SCHULTE, W. **Exception safety for C#.** In *SEFM 2004*—Second International Conference on Software Engineering and Formal Methods, pages 218–227. IEEE, September 2004.

SACRAMENTO, P., CABRAL, B., MARQUES, P. **Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?** In Second Edition of the International Conference on Innovative Views of .NET Technologies (IVNET'06).

SCHAEFER, C. F., BUNDY, G. N. **Static analysis of exception handling in Ada.** Softw. Pract. Exper., 23(10):pp. 1157–1174. 1993.

SHAH, H. CARSTEN, G., HARROLD, M. **Visualization of exception handling constructs to support program understanding**. In Proceedings of the 4th ACM symposium on Software visualization (SoftVis '08). ACM, New York, NY, USA, 19-28.

SINHA, S., ORSO, A., HARROLD, M. J. **Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow**. In ICSE ’04: Proceedings of the 26th International Conference on Software Engineering, pp. 336-345. IEEE Computer Society, Washington, DC, USA. 2004.

SLACK, N.; CHAMBERS, S., JOHNSTON, R. **Administração da produção.** 2ª ed., São Paulo, Editora Atlas, 2002.

TCU – TRIBUNADO DE CONTAS DA UNIÃO. **Auditoria no sistema de tratamento e visualização Radar X-4000**. Relator Ministro Benjamin Zymler. Brasília. Secretaria de Fiscalização de Tecnologia da Informação, 2008.

YEMINI, S., BERRY, D. **A Modular Verifiable Exception Handling Mechanism.** ACM Transactions on Programming Languages and Systems,7(2):pp. 214–243. 1985

APÊNDICE

Nesta seção são listados os artigos publicados pelo autor, como resultado da pesquisa realizada para construção desta dissertação, respectivamente: *eFlowMining: An Exception-Flow Analysis Tool for .NET Applications* e *Visualizando a Evolução do Comportamento Excepcional em Aplicações Multi-linguagem com eFlowMining.*